**Introduction to ASP.NET Core MVC Framework**

**Introduction to ASP.NET Core MVC Framework**

In this article, I am going to give you a brief introduction to**ASP.NET Core MVC** Framework. Please read our previous article where we discussed [**Developer Exception Page Middleware Components**](https://dotnettutorials.net/lesson/developer-exception-page-middleware-asp-net-core/) in ASP.NET Core Application. As part of this article, we are going to discuss the following pointers.

1. **What is MVC?**
2. **How MVC Design Pattern Works?**
3. **Understanding Model, View, and Controller.**
4. **Where the MVC Design Pattern is used in the real-time three-layer application?**
5. **What is ASP.NET Core MVC?**

**What is MVC?**

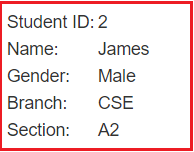
MVC stands for Model View and Controller. It is an architectural design pattern that means this design pattern is used at the architecture level of an application. So, the point that you need to remember is MVC is not a programming language, MVC is not a Framework, it is a design pattern. When we design an application, first we create the architecture of that application, and MVC plays an important role in the architecture of that particular application.

MVC Design Pattern is basically used to develop interactive applications. An interactive application is an application where there is user interaction involved and based on the user interaction some event handling occurred. The most important point that you need to remember is, it is not only used for developing web-based applications but also we can use this MVC design pattern to develop the Desktop or mobile-based application.

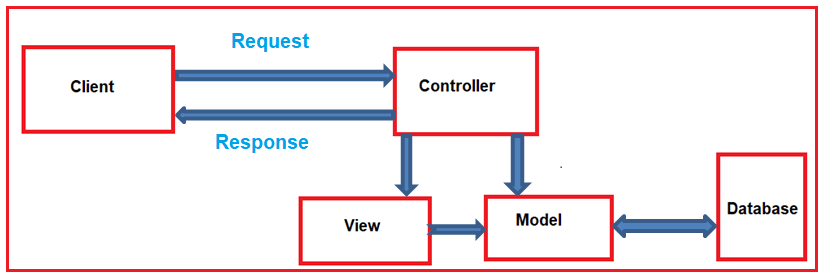
The MVC (Model-View-Controller) design pattern was introduced in the 1970s which divides an application into 3 major components. They are Model, View, and Controller. The main objective of the MVC design pattern is the separation of concerns. It means the domain model and business logic are separated from the user interface (i.e. view). As a result, maintaining and testing the application becomes simpler and easier.

**How does MVC Design Pattern work in ASP.NET Core?**

Let us see an example to understand how the MVC pattern works in the ASP.NET Core MVC application. For example, we want to design an application, where we need to display the student details on a web page as shown below.



So, when we issue a request something like “**http://dotnettutorials.net/student/details/2**” from a web browser then the following things are happening in order to handle the request.



The controller is the component in the MVC design pattern, who actually handles the incoming request. In order to handle the request, the controller components do several things are as follows. The controller component creates the model that is required by a view. The model is the component in the MVC design pattern which basically contains classes that are used to store the domain data or you can say business data.

In the MVC design pattern, the Model component also contains the required logic in order to retrieve the data from a database. Once the model created by the controller, then the controller selects a view to render the domain data or model data. While selecting a view, it is also the responsibility of the controller to pass the model data.

In the MVC design pattern, the only responsibility of view is to render the model data. So, in MVC, the view is the component whose responsibility is to generate the necessary HTML in order to render the model data. Once the HTML is generated by the view, then that HTML is then sent to the client over the network, who initially made the request.

So, the three major components of an ASP.NET Core MVC Application are Model, View, and Controller. Let us discuss each of these components of the MVC design pattern in detail.

**Model:**

The Model is the component in the MVC Design pattern which is used to manage that data i.e. state of the application in memory. The Model represents a set of classes that are used to describe the application’s validation logic, business logic, and data access logic. So in our example, the model consists of Student class and the StudentBusinessLayer class.

**public** **class** Student

**{**

**public** **int** StudentID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**}**

**public** **class** StudentBusinessLayer

**{**

**public** IEnumerable**<**Student**>** GetAll**()**

**{**

//logic to return all employees

**}**

**public** Student GetById**(int** StudentID**)**

**{**

//logic to return an employee by employeeId

Student student = new Student**()**

**{**

StudentID = StudentID,

Name = "James",

Gender = "Male",

Branch = "CSE",

Section = "A2",

**}**;

**return** student;

**}**

**public** **void** Insert**(**Student student**)**

**{**

//logic to insert an student

**}**

**public** **void** Update**(**Student student**)**

**{**

//logic to Update an student

**}**

**public** **void** Delete**(int** StudentID**)**

**{**

//logic to Delete an student

**}**

**}**

Here, in our example, we use the Student class to hold the student data in memory. The StudentBusinessLayer class is used to manage the student data i.e. going to perform the CRUD operation.

So, in short, we can say that a Model in MVC design pattern contains a set of classes that is used to represent the data and also contains the logic to manage those data. In our example, the Student class is the class that is used to represent the data. The StudentBusinessLayer class is the class that is used to manage the Student data.

**View:**

The view component in the MVC Design pattern is used to contain the logic to represent the model data as a user interface with which the end-user can interact. Basically, the view is used to render the domain data (i.e. business data) which is provided to it by the controller.

For example, we want to display Student data in a web page. In the following example, the Student model carried the student data to the view. As already discussed, the one and only responsibility of the view is to render that student data. The following code does the same thing.

@model ASPCoreApplication.Models.Student

**<html>**

**<head>**

**<title>**Student Details**</title>**

**</head>**

**<body>**

**<br/>**

**<br/>**

**<table>**

**<tr>**

**<td>**Student ID: **</td>**

**<td>**@Model.StudentID**</td>**

**</tr>**

**<tr>**

**<td>**Name: **</td>**

**<td>**@Model.Name**</td>**

**</tr>**

**<tr>**

**<td>**Gender: **</td>**

**<td>**@Model.Gender **</td>**

**</tr>**

**<tr>**

**<td>**Branch: **</td>**

**<td>**@Model.Branch**</td>**

**</tr>**

**<tr>**

**<td>**Section: **</td>**

**<td>**@Model.Section **</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

**Controller:**

A Controller is a .cs (for C# language) file which has some methods called Action Methods. When a request comes on the controller, it is the action method of the controller which will handle those requests.

The Controller is the component in an MVC application that is used to handle the incoming HTTP Request and based on the user action, the respective controller will work with the model and view and then sends the response back to the user who initially made the request. So, it is the one that will interact with both the models and views to control the flow of application execution. In our example, when the user issued a request the following URL

**http://dotnettutorials.net/student/details/2**

Then that request is mapped to the Details action method of the Student Controller. How it will map to the Details action method of the Student Controller that will discuss in our upcoming articles.

**public** **class** StudentController : Controller

**{**

**public** ActionResult Details**(int** studentId**)**

**{**

StudentBusinessLayer studentBL = new StudentBusinessLayer**()**;

Student studentDetail = studentBL.GetById**(**studentId**)**;

**return** View**(**studentDetail**)**;

**}**

**}**

As you can see in the example, the Student Controller creates the Student object within the Details action method. So, here the Student is the Model. To fetch the Student data from the database, the controller uses the StudentBusinessLayer class.

Once the controller creates the Student model with the necessary student data, then it passes that Student model to the Details view. The  Details view then generates the necessary HTML in order to present the Student data. Once the HTML is generated, then this HTML is sent to the client over the network who initially made the request.

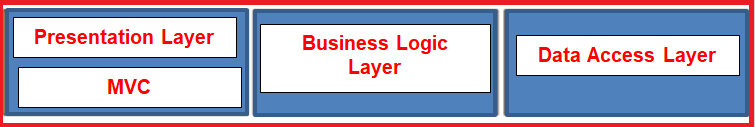
**Note:** In the MVC design pattern both the Controller and View depend on the Model. But the Model never depends on either view or controller. This is one of the main reasons for the separation of concerns. This separation of concerns allows us to build the model and test independently of the visual presentation.

**Where MVC is used in the real-time three-layer application?**

In general, a real-time application may consist of the following layers

1. **Presentation Layer:** This layer is responsible for interacting with the user.
2. **Business Layer:** This layer is responsible for implementing the core business logic of the application.
3. **Data Access Layer:** This layer is responsible for interacting with the database to perform the CRUD operations.

The MVC design pattern is basically used to implement the Presentation Layer of the application. Please have a look at the following diagram.



**What is ASP.NET Core MVC?**

The ASP.NET Core MVC is a lightweight, open-source, highly testable presentation framework that is used for building web apps and Web APIs using the Model-View-Controller (MVC) design pattern. So, the point that you need to remember is, MVC is a design pattern and ASP.NET Core MVC is the framework that is based on MVC Design Pattern.

The ASP.NET Core MVC Framework provides us with a patterns-based way to develop dynamic websites and web apps with a clean separation of concerns. This ASP.NET Core MVC framework provides us the full control over the mark-up. It also supports for Test-Driven Development and also uses the latest web standards.

# How to Set up MVC in ASP.NET Core

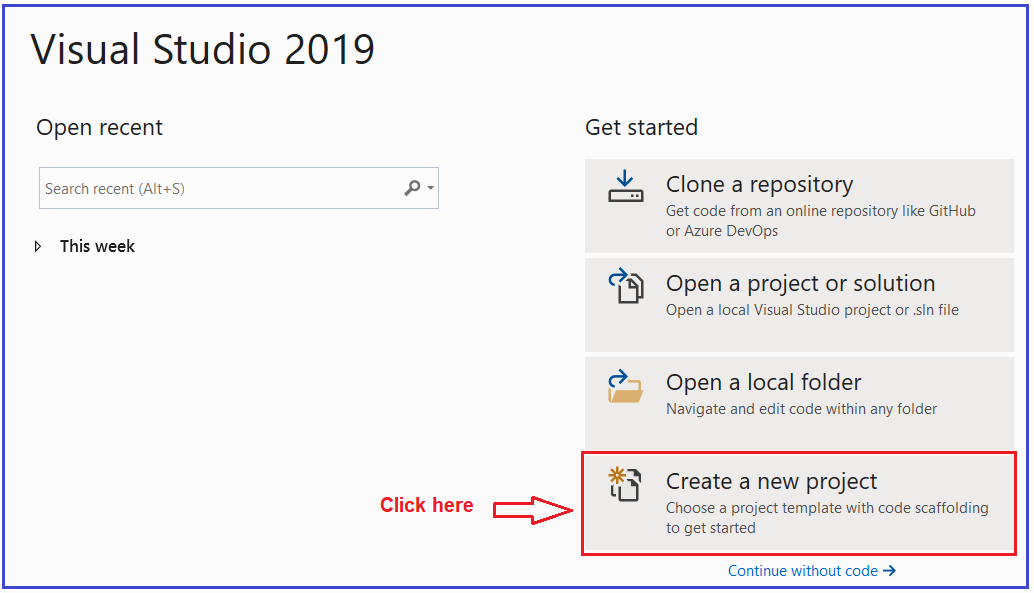
## ****How to Set up MVC in ASP.NET Core Application****

In this article, I am going to discuss **How to Set up MVC in ASP.NET Core Application** step by step. Please read our previous article before proceeding to this article where we discussed the [**Basics of ASP.NET Core MVC**](https://dotnettutorials.net/lesson/introduction-asp-net-core-mvc/) Application.

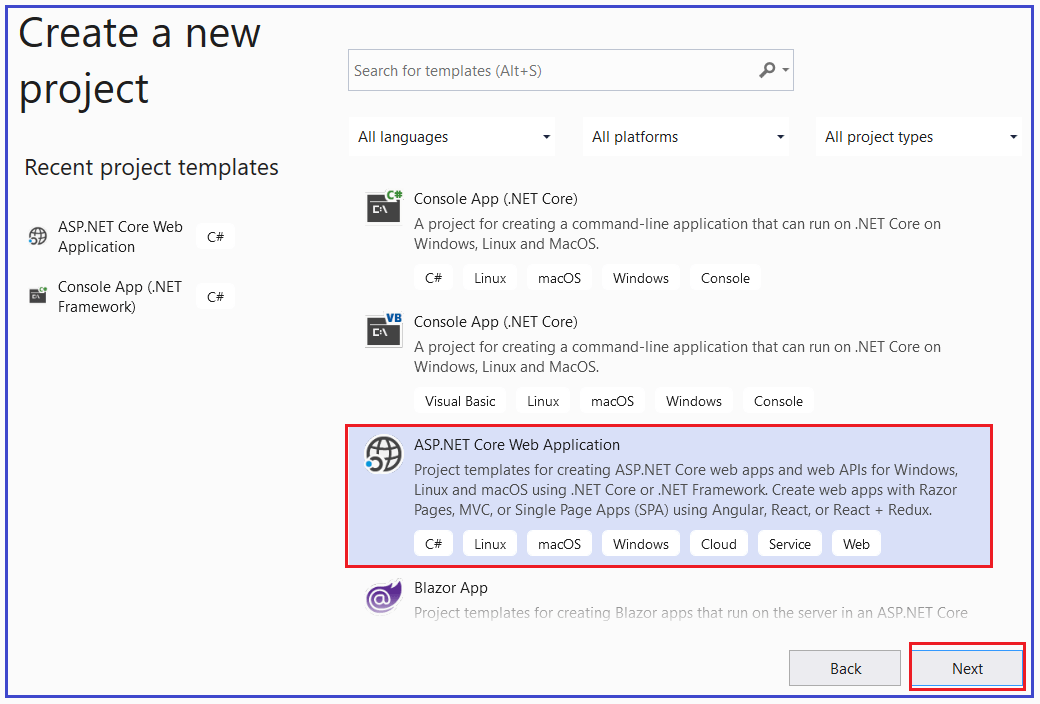
It is possible in ASP.NET Core to build an entire application using only the asp.net core middlewares. But the ASP.NET Core MVC framework provides us the features that we can use to create HTML pages and HTTP-based APIs easily. So, here we will see how to set up MVC in ASP.NET Core Application. To do so, let us first create an Empty ASP.NET Core Web Application. Later on this ASP.NET Core MVC article series, we will see examples using the Web Application (Model-View-Controller) template.

##### ****Creating a new ASP.NET Core Application:****

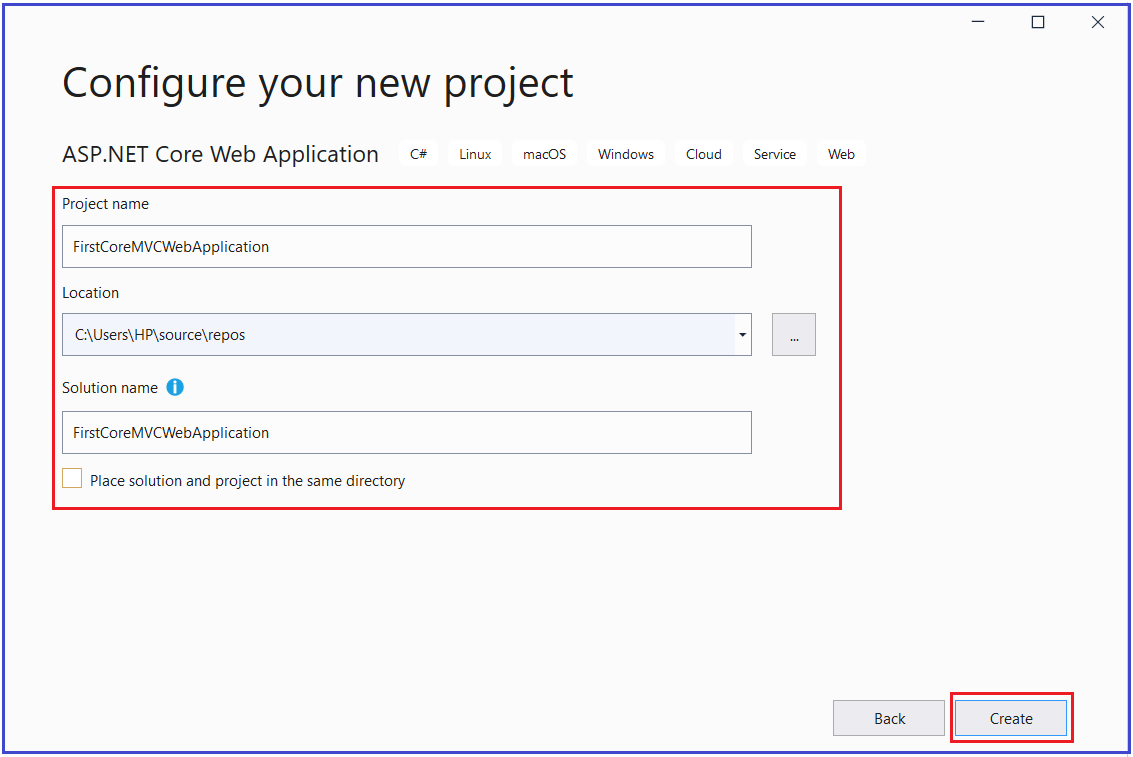
To create a new Empty ASP.NET Core Web Application, First, open Visual Studio 2019 and then click on the Create a new project tab as shown in the below image.



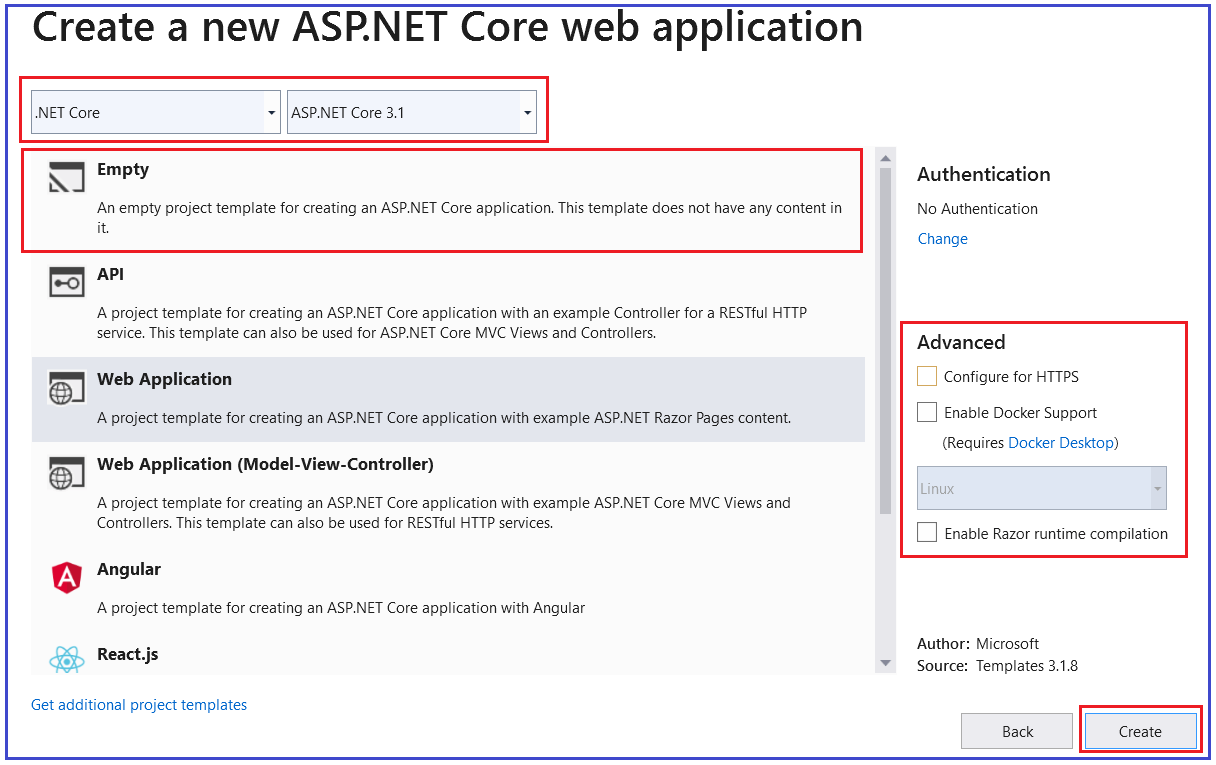
Once you click on the Create a new project tab, it will open the Create a new project window. From this window, you need to select the **ASP.NET Core Web Application** template and then click on the Next button as shown in the below image.



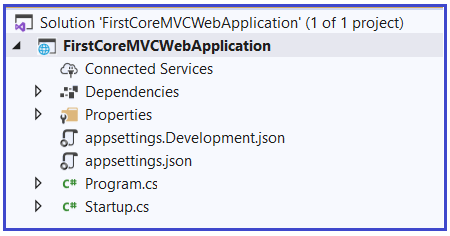
Once you click on the Next button, it will open the Configure Your New Project window. Here, you need to provide the necessary information to create a new project. First, give an appropriate name for your project (**FirstCoreMVCWebApplication**), set the location where you want to create this project, the solution name for the ASP.NET Core Web application. And finally, click on the Create button as shown in the image below.



Once you click on the create button, it will open the Create a new ASP.NET Core Web Application window where you can select the project template i.e. which type of project you want to create. As we are going to do everything manually and also from the scratch, so, select the Empty Project template and uncheck all the checkboxes from the Advanced section and finally click on the Create button as shown in the below image.



That’s it. Once you click on the Create Button, the project is going to be created with the Empty template with the following folder and file structure.



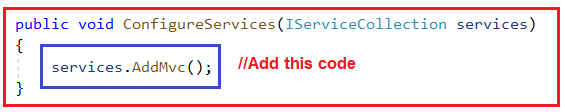
The Empty template by default does not include the setup for MVC. Now let us see how to set up MVC in asp.net core application.

#### ****Setup MVC in ASP.NET Core Application:****

There are two simple steps required to set up MVC in ASP.NET Core Application.

##### ****Adding MVC Service to the Dependency Injection Container:****

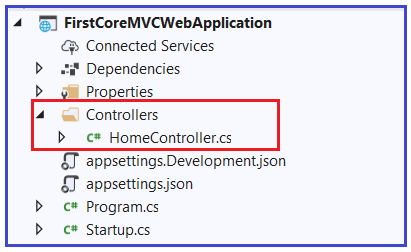
First, we need to add the required MVC services to the dependency injection container. To do so you need to modify the **ConfigureServices()**method of the Startup class which is present within the **Startup.cs** class file as shown below. The following piece of code will include all the required services which are required to develop ASP.NET core MVC application. Once you add this, you can use Models, Controllers, Views, and many other features in your ASP.NET Core Application.



**Note:** Along with **AddMVC()** method, we also have **AddControllersWithViews()** method. In the next article, we will discuss these two methods in detail as well as we will also discuss the difference between these two methods and when to use one over another.

##### ****Adding Controller in ASP.NET Core Application:****

In ASP.NET Core MVC application, all the Controllers should be present within a specific folder called Controllers. So first we need to add the **Controllers** folder within the root project folder. Once you add the Controllers folder then add a new class file with the name **HomeController** within the Controllers folder. Once you add the HomeController class, your project folder structure should be as shown below.



Now open the HomeController class and then copy and paste the following code in it. In order to make a class as a controller in ASP.NET Core MVC, that class needs to be inherited from the **Controller** base class. So, you can see in the below code our controller i.e. HomeController is inherited from the Controller class. This Controller class belongs to **Microsoft.AspNetCore.Mvc** namespace.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** string Index**()**

**{**

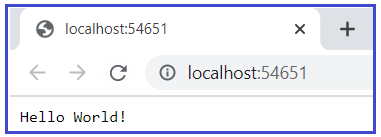
**return** "This is Index action from MVC Controller";

**}**

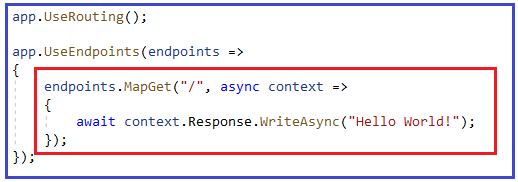
**}**

**}**

With the above changes in place, now run the application and see the output as shown in the below image.



We are not getting the output from the Index action method of the HomeController. The above output is coming from the below code which you can find in the Configure method of Startup class.



##### ****How to configure our Index Action Method of HomeController?****

We need to tell to the MVC framework to use the Index action method of our Home Controller as the default route. To do so, we need to add the **required MVC middleware** to the application request processing pipeline. So, modify the Configure Method of Startup class as shown below. The MapDefaultControllerRoute() method add the MVC middleware to the request processing pipeline.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

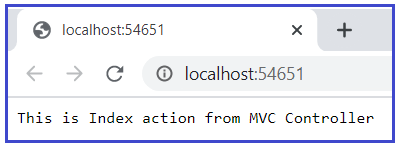
//Configuring the MVC middleware to the request processing pipeline

endpoints.MapDefaultControllerRoute**()**;

**})**;

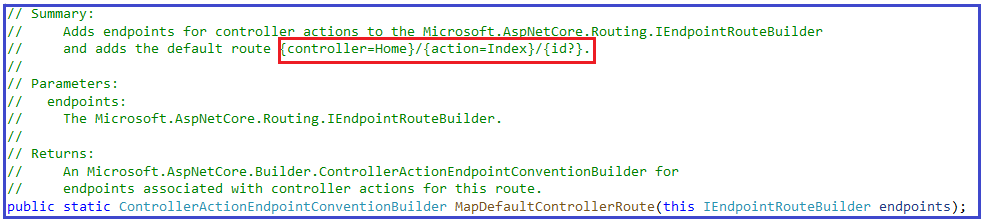
**}**

Now, run the application and you should get the output as expected as shown in the below image.



##### ****Understanding MapDefaultControllerRoute() middleware****

Let have a look at the definition of the MapDefaultControllerRoute**()** middleware.



As shown in the above image, the default controller is Home and the default action method is Index for our application. This is the reason why when we run the application, the Index action method of the Home Controller handles the request. But if you want then you can also change this default behavior and that we will discuss in our upcoming articles.

So, in short, to Setup MVC in asp.net core application, first we need to add the required MVC services to the dependency injection container and secondly, we need to configure the MVC middleware in the request processing pipeline.

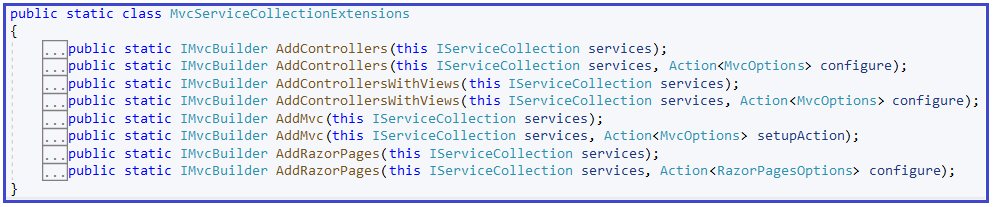
**AddController vs AddMvc vs AddControllersWithViews vs AddRazorPages**

**AddController vs AddMvc vs AddControllersWithViews vs AddRazorPages**

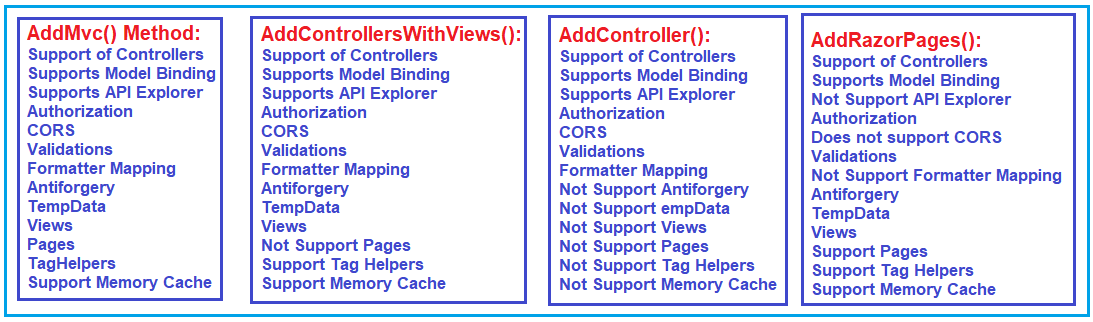
In this article, I am going to discuss the **AddController() vs AddMvc() vs AddControllersWithViews() vs AddRazorPages()** method in ASP.NET Core application. We will also discuss when to use what methods in ASP.NET Core. I strongly recommended you read our previous article before proceeding to this article where we discussed [**how to set MVC using the AddMvc method in ASP.NET Core**](https://dotnettutorials.net/lesson/setup-mvc-asp-net-core-application/) application.

**Different methods available in ASP.NET Core:**

If you go to the definition of AddMvc() extension method, you will see along with AddMvc() method, there are also other methods (AddController(), AddControllersWithViews(), and AddRazorPages()) available as shown in the below image. All these methods are implemented as an extension method on the IServiceCollection interface.



Let us discuss what all these methods are and what features they provide in detail one by one.



**Points to Remember:**

1. Support of controller is available for all the methods. So, if you need only a controller then you can use any of the methods.
2. The model binding feature is also available for all the methods. Model binding is used to map the incoming data to the controller action methods.
3. Except for the AddRazorPages method, all other methods support the API Explorer feature. The API Explorer has used the list of all the available APIs in your application.
4. Authorization is available for all four methods. Authorization is basically used to provide the security features,
5. Again, except for the AddRazorPages method, all other methods support CORS. CORS is basically a feature that allows CROSS domain call. That means from other domains they can access your method using jQuery AJAX.
6. The validation feature is supported by all the methods. Validation is basically used to validate the HTTP Request data. In .NET Core Application, we can implement validation using a concept called Data Annotation.
7. Except for the AddRazorPages method, all other methods support the Formatter Mapping feature. The Formatter Mapping feature is basically used to format the output of your action method such as JSON or XML, etc.
8. Antiforgery, TempData, and Views features are not available in the AddControllers method.
9. The Pages are available only with AddMVC and AddRazorPages method.
10. TagHelpers are not available in the AddControllers method and available for rest three methods.
11. The memory Cache feature is also not available in the AddControllers method but available with the rest three methods.

**Which method to use for our application?**

This is depending on which type of application you want to create.

1. If you want to create a Web API application where there are no views, then you need to use AddControllers() extension method.
2. If you want to work with the Razor Page application, then you need to use the AddRazorPages() extension method into your ConfigureService method of Startup class.
3. If you want to develop a Model View Controller i.e. MVC application then you need to use AddControllersWithViews() method. Further, if you want Pages features into your MVC application, then you need to use the AddMVC method.

**AddMvc:** This method has all the features. So, you can any type of application (Web API, MVC, and Razor Pages) using this AddMVC method.

**Note:** Adding AddMvc() method will add extra features even though which are not required to your application which might impact the performance of the application.

**Models in ASP.NET Core MVC**

**Models in ASP.NET Core MVC Application**

In this article, I am going to discuss the **Models in ASP.NET Core MVC** application with Examples. Please read our previous article where we discussed [**AddController() vs AddMvc() vs AddControllersWithViews() vs AddRazorPages() method in ASP.NET Core**](https://dotnettutorials.net/lesson/difference-between-addmvc-and-addmvccore-method/) Application. We are going to work with the same example that we created in our [**How to set up MVC in ASP.NET Core**](https://dotnettutorials.net/lesson/setup-mvc-asp-net-core-application/) article.

**What is a Model in ASP.NET Core MVC?**

A model is a class with .cs (for C#) as an extension having both properties and methods. Models are used to set or get the data. If your application does not have data, then there is no need for a model. If your application has data, then you need a model.

**What is the Role of Models in ASP.NET Core MVC?**

The Models in ASP.NET Core MVC contains a set of classes that are used to represent the domain data (you can also say the business data) as well as it also contains logic to manage the domain/business data. So, in simple words, we can say that the models in ASP.NET Core MVC Application are used to manage the data i.e. the state of the application in memory.

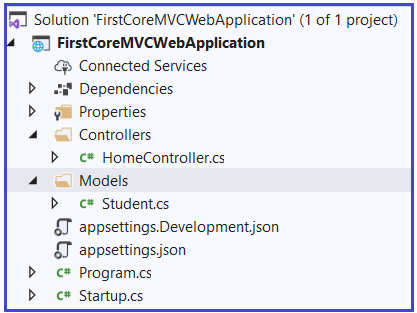
If you are working with any Web Application that is based on MVC Design Pattern, then in that MVC Application, three things are common i.e. Model, View, and Controller. The Controllers are used to manage the overall flow of the application. Models are responsible for the data and these data are used on Views. Views are basically the HTML pages that get rendered into the browser of the client. In the browser, we generally perform two operations. First, we display the data to the user and secondly, we get the data from the user. And for both these operations models are used.

**Note:** It is not mandatory, but it is a good programming practice to store all the model classes within the Models folder. Even though you can also create a class library project and put all the model classes in that class library project and refer that class library project in your application and we will discuss this as we progress in this course.

When you create a new ASP.NET Core Application using MVC Template, then by default all the model classes are created inside the Models folder. And we are also going to follow this naming convention. Let us see how to create and work with models in ASP.NET Core MVC.

**Adding Models Folder in ASP.NET Core Application:**

Right-click on your project, then select **add => new folder** option from the context menu which will add a new folder. Then rename the folder name as **Models**. Here we want to create a model for displaying the student data. So, create a class file with the name **Student.cs** within the **Models** folder. Once you create the Student model then the folder structure of your application should looks as shown below.



**Now open the Student.cs class file and then copy and paste the following code.**

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

This is our student model which is going to store the student data in memory. As we already discussed, the model in ASP.NET Core MVC Application also contains business logic to manage the data. So, in our example, to manage the student data i.e. to perform the CRUD operation on the student data we are going to use the following IStudentRepository interface.

**Creating IStudentRepository interface:**

Right-click on the Models folder and then add an interface with the name **IStudentRepository.cs**. Once you create the interface then copy and paste the following code in it.

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **interface** IStudentRepository

**{**

Student GetStudentById**(int** StudentId**)**;

**}**

**}**

As you can see, we created the above interface with one method i.e. GetStudentById() method which will retrieve the student details by the student id.

**Creating TestStudentRepository class:**

Let us create an implementation class for the above **IStudentRepository** interface. In our upcoming article, we will discuss how to retrieve the student details from a database. But for this demo, lets hardcoded the student details. So, create a class file with the name TestStudentRepository.cs within the Models folder and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **class** TestStudentRepository : IStudentRepository

**{**

**public** List**<**Student**>** DataSource**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 102, Name = "Smith", Branch = "ETC", Section = "B", Gender = "Male" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 104, Name = "Sara", Branch = "CSE", Section = "A", Gender = "Female" **}**,

new Student**()** **{** StudentId = 105, Name = "Pam", Branch = "ETC", Section = "B", Gender = "Female" **}**

**}**;

**}**

**public** Student GetStudentById**(int** StudentId**)**

**{**

**return** DataSource**()**.FirstOrDefault**(**e =**>** e.StudentId == StudentId**)**;

**}**

**}**

**}**

**Modify HomeController:**

We already created a Controller with the name HomeController within the Controller Folders. If you have not created yet then add a class file with the HomeController within the Controllers folder. And then modify the HomeController as shown below to use the TestStudentRepository to retrieve the student details. The Student and TestStudentRepository are present in a separate namespace, so you need to include the namespaces as well.

**using** *FirstCoreMVCWebApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** JsonResult GetStudentDetails**(int** Id**)**

**{**

TestStudentRepository repository = new TestStudentRepository**()**;

Student studentDetails = repository.GetStudentById**(**Id**)**;

**return** Json**(**studentDetails**)**;

**}**

**}**

**}**

If you are directly come to this article, without reading our previous article, then please modify the Startup class as shown below where we register the MVC service to the built-in dependency Injection Container as well as we add MVC Middleware to the application request processing pipeline.

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreMVCWebApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddControllersWithViews**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapDefaultControllerRoute**()**;

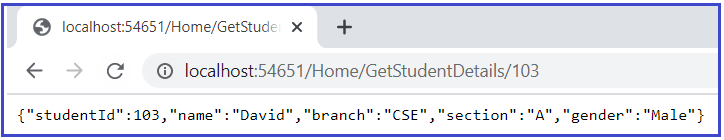
**})**;

**}**

**}**

**}**

Now run the application and navigate to [http://localhost:<portnumber>/Home/GetStudentDetails/103](http://localhost:%3cportnumber%3e/Home/GetStudentDetails/103) URL and you will see the student data in JSON format as expected in the browser as shown in the below image.



The way we implemented the GetStudentDetails method of Home Controller is not loosely coupled. That means tomorrow if the implementation class of the IStudentRepository is changed then we need to change the code in the Home Controller class as both are tightly coupled. We can overcome this problem by implementing a dependency injection design pattern.

# ASP.NET Core Dependency Injection

## ****ASP.NET Core Dependency Injection with Example****

In this article, I am going to discuss the **ASP.NET Core Dependency Injection** with an example. Please read our previous article before proceeding to this article where we discussed [**Models in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/model-asp-net-core-mvc/) application. The Dependency Injection Design Pattern is one of the most used design Patterns in real-time applications. But the good thing is that ASP.NET Core Provides inbuilt support for Dependency Injection. As part of this article, we are going to discuss the following pointers in details.

1. **Understanding the need for ASP.NET Core Dependency Injection**
2. **What is Dependency Injection?**
3. **How to register a Service with ASP.NET Core Dependency Injection Container?**
4. **What are the different methods ASP.NET Core Provides to register a service with Dependency Injection Contains?**
5. **Understanding the Singleton, Scoped, and Transient Methods**
6. **What are the advantages of using Dependency Injection?**

##### ****Understanding the need for ASP.NET Core Dependency Injection****

Let us understand the need for Dependency Injection in ASP.NET Core Application with an example. First, create a new ASP.NET Core Application with the name “**FirstCoreMVCWebApplication**” with Empty project template.

##### ****Adding Models:****

Once you created the Project with the Empty Project template, then let’s add our models to our application. To do so, first create a folder with the name **Models**. Within the Models folder, let us add a class file with the name **Student.cs** and this Student class is going to be our model for this application. Then open the **Student.cs** class file and copy and paste the following code in it.

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

##### ****Creating Service Interface:****

Then create an interface with the name **IStudentRepository.cs**within the Models folder. This interface is going to declare the list of methods that we can perform on the student data. So, open IStudentRepository.cs and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **interface** IStudentRepository

**{**

Student GetStudentById**(int** StudentId**)**;

List**<**Student**>** GetAllStudent**()**;

**}**

**}**

##### ****Creating Service Implementation:****

Create a class file with the name **TestStudentRepository.cs** within the same Models folder. Then open the TestStudentRepository.cs file and copy-paste the following code in it. This class implements the **IStudentRepository**interface by providing the implementation of the two methods declared in that interface.

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **class** TestStudentRepository : IStudentRepository

**{**

**public** List**<**Student**>** DataSource**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 102, Name = "Smith", Branch = "ETC", Section = "B", Gender = "Male" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 104, Name = "Sara", Branch = "CSE", Section = "A", Gender = "Female" **}**,

new Student**()** **{** StudentId = 105, Name = "Pam", Branch = "ETC", Section = "B", Gender = "Female" **}**

**}**;

**}**

**public** Student GetStudentById**(int** StudentId**)**

**{**

**return** DataSource**()**.FirstOrDefault**(**e =**>** e.StudentId == StudentId**)**;

**}**

**public** List**<**Student**>** GetAllStudent**()**

**{**

**return** DataSource**()**;

**}**

**}**

**}**

##### ****Startup.cs:****

In the Startup class initially, we need to do two things. First, we need to configure the required MVC service to the IoC Container, and then we need to add the MVC Middleware to the request processing pipeline. So, modify the Startup class as shown below.

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreMVCWebApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddControllersWithViews**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapDefaultControllerRoute**()**;

**})**;

**}**

**}**

**}**

##### ****Without Dependency Injection:****

Create a folder with the name Controllers in your project. Then add a class file with the name **HomeController.cs** within the Controllers folder. Then open the **HomeController.cs** file and copy-paste the following code in it.

**using** *FirstCoreMVCWebApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** JsonResult Index**()**

**{**

TestStudentRepository repository = new TestStudentRepository**()**;

List**<**Student**>** allStudentDetails = repository.GetAllStudent**()**;

**return** Json**(**allStudentDetails**)**;

**}**

**public** JsonResult GetStudentDetails**(int** Id**)**

**{**

TestStudentRepository repository = new TestStudentRepository**()**;

Student studentDetails = repository.GetStudentById**(**Id**)**;

**return** Json**(**studentDetails**)**;

**}**

**}**

**}**

With the above changes in place, now run the application and check the above two methods and it should work as expected. Let us first understand what is the problem in the above implementation and how we can overcome this by using the dependency injection design pattern in the ASP.NET Core application.

##### ****What is the Problem in the above implementation?****

As you can see in the above HomeController class, in order to get student data, the HomeController class depends on the TestStudentRepository class. Here within the HomeController class, we create an instance of TestStudentRepository class and then invoke the GetStudentById()and GetAllStudent method as per our requirement. This is tight coupling because the HomeController class is now tightly coupled with the TestStudentRepository class.

Tomorrow if the implementation class of the IStudentRepository is changed then we also need to change the code in the HomeController class as they both are tightly coupled. We can overcome this problem by implementing the dependency injection design pattern in ASP.NET Core Application.

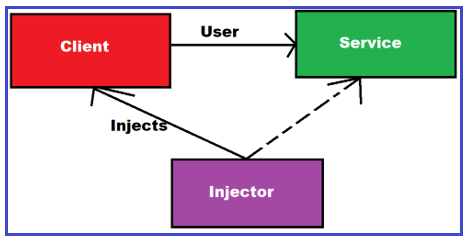
##### ****What is Dependency Injection (DI) Design Pattern?****

The Dependency Injection a process of injecting the object of a class into a class that depends on it. The Dependency Injection is the most commonly used design pattern nowadays to remove the dependencies between the objects that allow us to develop loosely coupled software components.

Let us discuss the step by step procedure to implement dependency injection in ASP.NET Core MVC application.

##### ****Dependency Injection in ASP.NET Core:****

The ASP.NET Core Framework is designed from scratch to support inbuilt support for Dependency Injection. The ASP.NET Core Framework injects objects of dependency classes through constructor or method by using a built-in IoC (Inversion of Control) container.



ASP.NET Core framework contains simple out-of-the-box IoC containers which do not have as many features as other third party IoC containers such as Unity, StructureMap, Castle Windsor, Ninject, etc. If you want more features such as auto-registration, scanning, interceptors, or decorators then you may replace the built-in IoC container with a third-party container.

The built-in container is represented by IServiceProvider implementation that supports constructor injection by default. The types (classes) managed by built-in IoC containers are called services.

##### ****Types of Services in ASP.NET Core:****

There are two types of services in ASP.NET Core. They are as follows:

1. **Framework Services:** Services that are a part of the ASP.NET Core framework such as IApplicationBuilder, IHostingEnvironment, ILoggerFactory, etc.
2. **Application Services:** The services (custom types or classes) which you as a programmer create for your application.

In order to let the IoC container automatically inject our application services, we first need to register them with the IoC container.

##### ****How to register a Service with ASP.NET Core Dependency Injection Container?****

We need to register a service with ASP.NET Core Dependency Injection Container within the ConfigureServices() method of the Startup class.

Before we discuss how to register a service with the Dependency Injection Container, it is important to understand the lifetime of service. When a class receives the dependency object through dependency injection, then whether the instance it receives is unique to that instance of the class or not depends on the lifetime of the service. Setting the lifetime of the dependency object determines how many times the dependency object needs to be created.

##### ****What are the different methods ASP.NET Core Provides to register a service with Dependency Injection Contains?****

The ASP.NET core provides 3 methods to register a service with the ASP.NET Core Dependency Injection container as follows. The method that we use to register a service will determine the lifetime of that service.

1. **Singleton**
2. **Transient**
3. **Scoped**

**Singleton:** In this case, the IoC container will create and share a single instance of a service object throughout the application’s lifetime.

**Transient:** In this case, the IoC container will create a new instance of the specified service type every time you ask for it.

**Scoped:** In this case, the IoC container will create an instance of the specified service type once per request and will be shared in a single request.

**Note:** The Built-in IoC container manages the lifetime of a registered service. It automatically disposes of a service instance based on the specified lifetime.

##### ****Registering the TestStudentRepository with ASP.NET Core Dependency Injection****

We need to configure the service instance within the ConfigureServices() method of the Startup class. The following code shows how to register a service with different lifetimes:

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Adding MVC Service. Framework Service

services.AddControllersWithViews**()**;

//Application Service

services.Add**(**new ServiceDescriptor**(**typeof**(**IStudentRepository**)**, new TestStudentRepository**()))**; // by default singleton

services.Add**(**new ServiceDescriptor**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**)**, ServiceLifetime.Singleton**))**; // singleton

services.Add**(**new ServiceDescriptor**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**)**, ServiceLifetime.Transient**))**; // Transient

services.Add**(**new ServiceDescriptor**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**)**, ServiceLifetime.Scoped**))**; // Scoped

**}**

##### ****Extension Methods for Registration****

ASP.NET Core framework includes extension methods for each types of lifetime; AddSingleton(), AddTransient() and AddScoped() methods for singleton, transient and scoped lifetime respectively. The following example shows the ways of registering types (service) using extension methods.

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Adding MVC Service. Framework Service

services.AddControllersWithViews**()**;

//Application Service

services.AddSingleton**<**IStudentRepository, TestStudentRepository**>()**;

services.AddSingleton**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

services.AddTransient**<**IStudentRepository, TestStudentRepository**>()**;

services.AddTransient**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

services.AddScoped**<**IStudentRepository, TestStudentRepository**>()**;

services.AddScoped**(**typeof**(**IStudentRepository**)**, typeof**(**TestStudentRepository**))**;

**}**

So, let us use the Single Instance of the service in this example. So, modify the ConfigureService method of the Startup class as shown below. Which method you want to use to register your application service to the built-in IoC Container is your personal preference. I am going to use the following.

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Adding MVC Service. Framework Service

services.AddControllersWithViews**()**;

//Application Service

services.AddSingleton**<**IStudentRepository, TestStudentRepository**>()**;

**}**

##### ****Constructor Injection in ASP.NET Core MVC Application****

Once we register the service, the IoC container automatically performs constructor injection if a service type is included as a parameter in a constructor. Let us modify the HomeController as shown below to use the Constructor dependency injection.

**using** *FirstCoreMVCWebApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

//Create a reference variable of IStudentRepository

**private** **readonly** IStudentRepository \_repository = **null**;

//Initialize the variable through constructor

**public** HomeController**(**IStudentRepository repository**)**

**{**

\_repository = repository;

**}**

**public** JsonResult Index**()**

**{**

List**<**Student**>** allStudentDetails = \_repository.GetAllStudent**()**;

**return** Json**(**allStudentDetails**)**;

**}**

**public** JsonResult GetStudentDetails**(int** Id**)**

**{**

Student studentDetails = \_repository.GetStudentById**(**Id**)**;

**return** Json**(**studentDetails**)**;

**}**

**}**

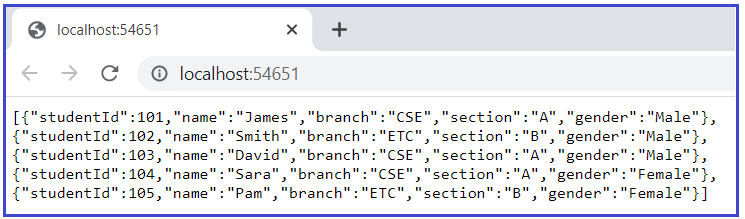
**}**

###### **Code Explanation:**

In the above example, the IoC container will automatically pass an instance of the **TestStudentRepository** to the constructor of HomeController. We don’t need to do anything else. An IoC container will create and dispose of an instance of the **IStudentRepository**based on the registered lifetime. As we are injecting the dependency object through a constructor, it is called as constructor dependency injection.

We created the \_**repository**variable as read-only which will ensure that once we injected the dependency object then that value can never be changed.

At this point, run the application and you should get the output as expected as shown in the below image.



##### ****Action Method Injection in ASP.NET Core Application****

Sometimes we may only need a dependency service type in a single action method. For this, use the **[FromServices]** attribute with the service type parameter in the method.

**using** *FirstCoreMVCWebApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** HomeController**()**

**{**

**}**

**public** JsonResult Index**([**FromServices**]** IStudentRepository repository**)**

**{**

List**<**Student**>** allStudentDetails = repository.GetAllStudent**()**;

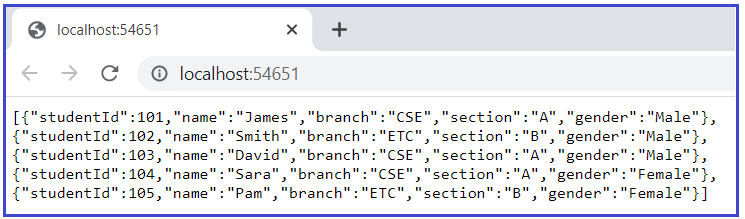
**return** Json**(**allStudentDetails**)**;

**}**

**}**

**}**

Run the application and you will get the expected output as shown below.



##### ****Property Injection****

The Built-in IoC container does not support property injection. You will have to use a third-party IoC container.

##### ****Get Services Manually****

It is not required to include dependency services in the constructor. We can access dependent services configured with built-in IoC containers manually using the RequestServices property of HttpContext as shown below.

**using** *FirstCoreMVCWebApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** JsonResult Index**()**

**{**

var services = this.HttpContext.RequestServices;

var \_repository = **(**IStudentRepository**)**services.GetService**(**typeof**(**IStudentRepository**))**;

List**<**Student**>** allStudentDetails = \_repository.GetAllStudent**()**;

**return** Json**(**allStudentDetails**)**;

**}**

**public** JsonResult GetStudentDetails**(int** Id**)**

**{**

var services = this.HttpContext.RequestServices;

var \_repository = **(**IStudentRepository**)**services.GetService**(**typeof**(**IStudentRepository**))**;

Student studentDetails = \_repository.GetStudentById**(**Id**)**;

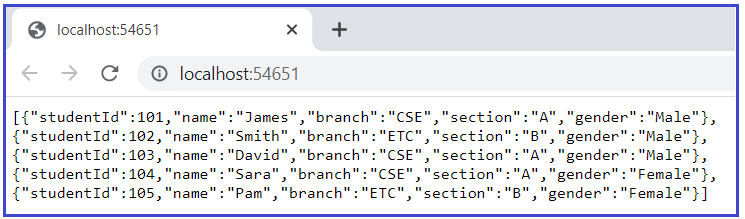
**return** Json**(**studentDetails**)**;

**}**

**}**

**}**

With the above changes in place, run the application and you should get the output as expected as shown in the below image.



**Note:**It is recommended to use constructor injection instead of getting it using RequestServices.

##### ****When to use what?****

In real-time applications, you need to register the components such as application-wide configuration as Singleton. The Database access classes like Entity Framework contexts are recommended to be registered as Scoped so that the connection can be re-used. If you want to run anything in parallel then it is better to register the component as Transient.

###### **So, in short:**

**AddSingleton():**When we use the AddSingleton() method to register a service, then it will create a singleton service. It means a single instance of that service is created and that singleton instance is shared among all the components of the application that require it. That singleton service is created when we requested for the first time.

**AddScoped():**Scoped means instance per request. When we use the AddScoped() method to register a service, then it will create a Scoped service. It means, an instance of the service is created once per each HTTP request and uses that instance in other calls of the same request.

**AddTransient():**When we use the AddTransient() method to register a service, then it will create a Transient service. It means a new instance of the specified service is created each time when it is requested and they are never shared.

##### ****What are the advantages of using ASP.NET Core Dependency Injection?****

The ASP.NET Core Dependency Injection allows us to develop loosely coupled software components. Using the ASP.NET Core Dependency Injection, it is very easy to swap with a different implementation of a component.

**Controllers in ASP.NET Core MVC**

**Controllers in ASP.NET Core MVC Application**

In this article, I am going to discuss the Controllers in **ASP.NET Core MVC Application** with an example. Please read our previous article before proceeding to this article where we discussed [**ASP.NET Core Dependency Injection**](https://dotnettutorials.net/lesson/asp-net-core-dependency-injection/) with an example. As part of this article, we are going to discuss the following pointers.

1. **What is a Controller?**
2. **Role of Controller**
3. **How to add Controller in ASP.NET Core Application?**
4. **What are Action Methods?**
5. **How to call the Action method of a Controller?**
6. **How to Pass Parameters in Action Methods?**
7. **When should we create a new controller?**
8. **How many controllers can we have in a single application?**

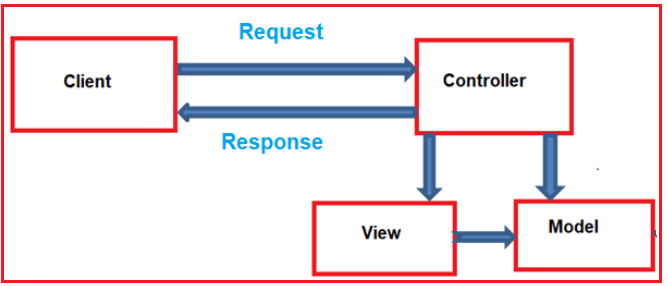
**What is a Controller in ASP.NET Core?**

A Controller is a special class in ASP.NET Core Application with .cs (for C# language) extension. By default, when you create a new ASP.NET Core Application using Model View Controller (MVC) template, then you can see the Controllers are residing in the Controllers folder.

The Controllers in MVC application logically group similar types of actions together. This aggregation of actions or grouping similar types of action together allows us to define sets of rules such as caching, routing, and authorization which is going to be applied collectively.

In ASP.NET Core MVC Application, the controller class should and must be inherited from the Controller base class.

When the client (browser) sends a request to the server, then that request first goes through the request processing pipeline. Once the request passes the request processing pipeline, it will hit the controller. Inside the controller, there are lots of methods (called action methods) actually handle that incoming HTTP Request. The action method inside the controller executes the business logic and prepared the response which is sent back to the client who initially made the request.



**Role of Controller:**

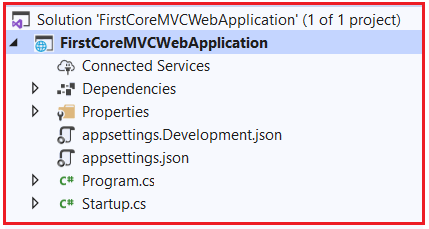
1. A Controller is used to group actions i.e. Action Methods.
2. The Controller is responsible to handle the incoming HTTP Request.
3. The Mapping of the HTTP Request is done using Routing. That is for a given HTTP Request, which action method of which controller is going to invoke is handled by the Routing Engine.’
4. Many important features such as Caching, Security, etc. can be applied to the controller.

**How to add Controllers in ASP.NET Core Application?**

If you create the ASP.NET Core Application using the MVC Project Template, then by default it will create a controller called HomeController within the Controllers folder. But if you create the ASP.NET Core Application with Empty Project template, then by default you will not find the Controllers folder in your project. As we are going to discuss everything from scratch, so we will create the ASP.NET Web Core Application with Empty Template and will add the Controllers folder and the Controllers manually.

**Step1: Creating Empty ASP.NET Core Web Application**

Create an ASP.NET Core Web Application with an Empty Project template. You can give any name to your application. I am giving the name for My Application as **FirstCoreMVCWebApplication.**The project will be created with the following folder structure and you can see there is no such folder called Controllers.

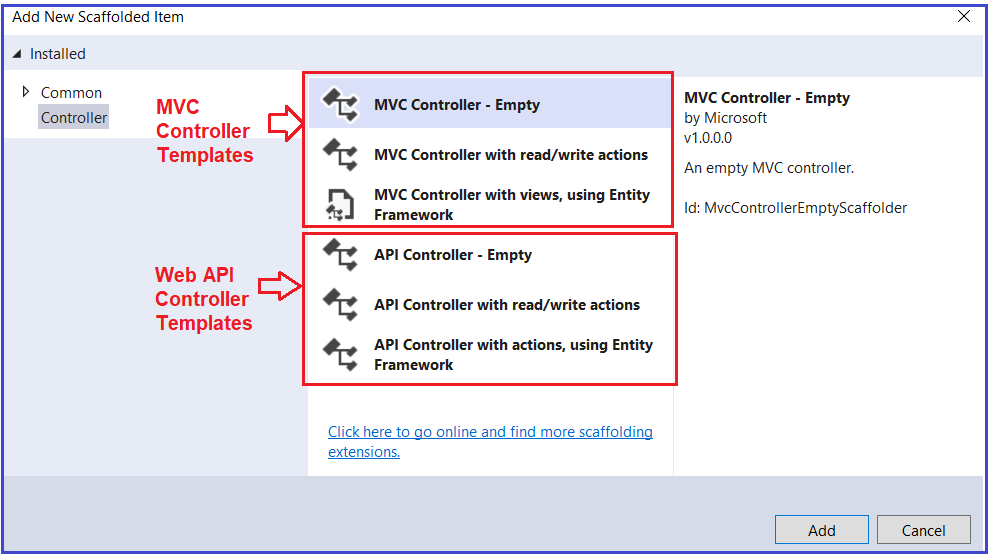


**Step2: Adding Controllers folder**

Once you create the project, next we need to add the Controllers folder where we will create our Controllers. To do so, right-click on the project and then select the add => new folder option from the context menu which will add a new folder in your project. Just rename the folder as Controllers.

**Step3: Adding Controller**

Once you create the Controllers folder, next we need to add a controller (StudentController) inside this Controllers folder. To do so, right-click on the Controller folder and select the Add => Controller option from the context menu which will open the Add Controller window as shown in the below image.



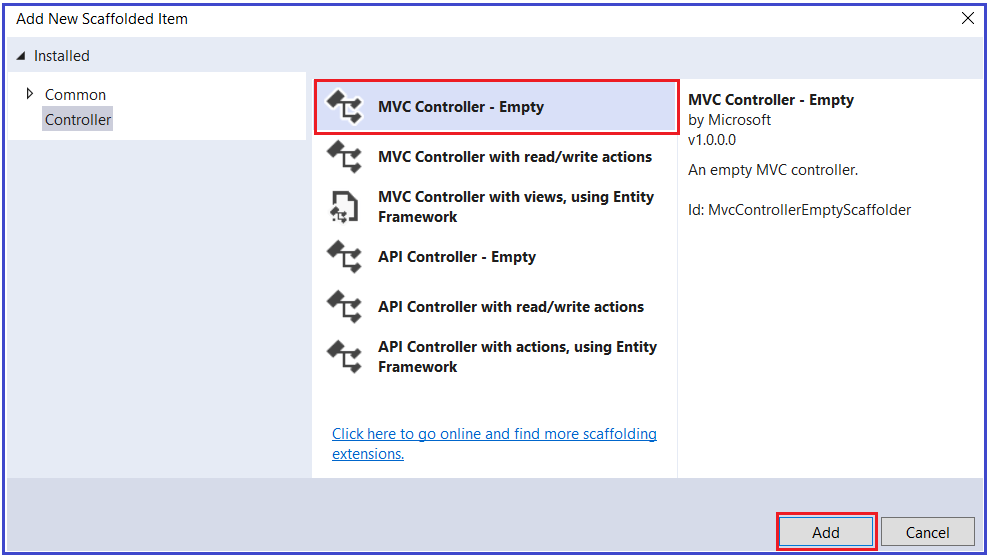
As you can see in the above image, we have three templates to create an MVC controller as well as three templates to create a Web API Controller. As we are interested MVC Application, so you can use any of the following three templates:

**MVC Controller – Empty**: It will create an Empty Controller.

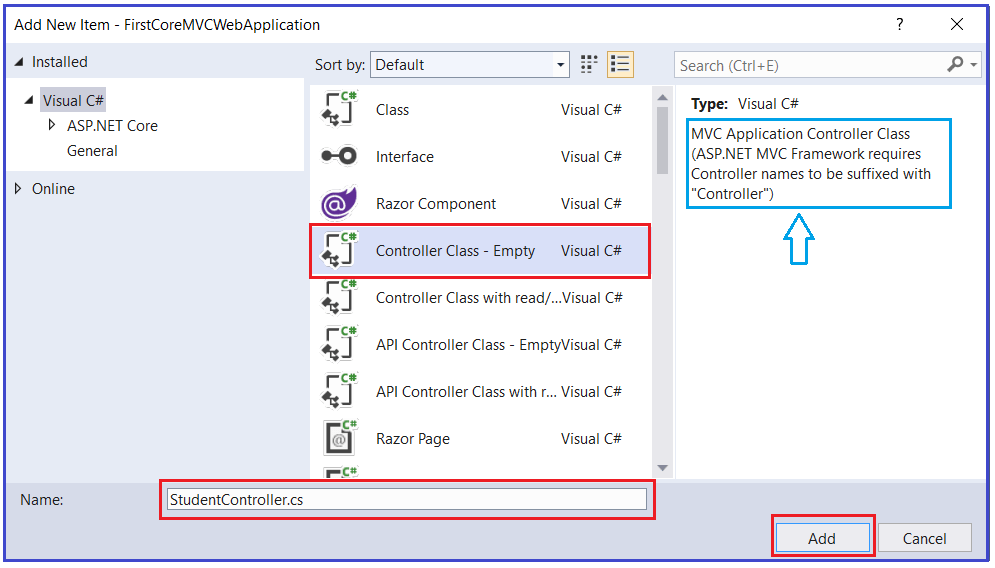
**MVC Controller with read/write actions**: This template will create the controller with five action methods to create, read, update, delete, and list entities.

**MVC Controller with views, using Entity Framework**: This template will create an MVC Controller with actions and Razor views to create, read, update, delete, and list entities using Entity framework.

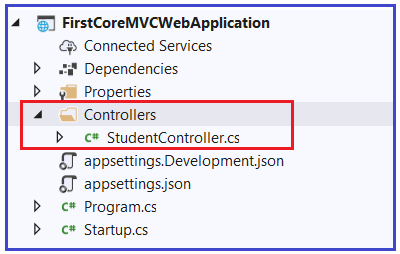
Here, we are going to create the MVC Controller with the Empty template. So, select the MVC Controller – Empty option and click on the Add button as shown in the below image.



Once you click on the Add button, it will open the below window where you need to select the **Controller Class – Empty** option and give a meaningful name to your controller. Here, I am giving the name as **StudentController** and click on the Add button. Here, the point that you need to remember is the Controller names should be suffixed with the word Controller.

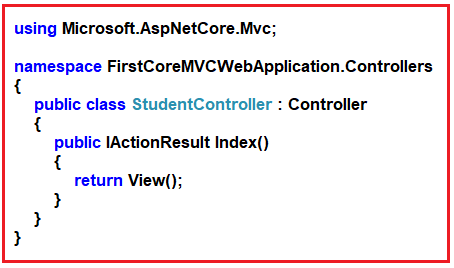


Once you click on the Add button then it will add StudentController within the Controllers folder as shown in the below image.

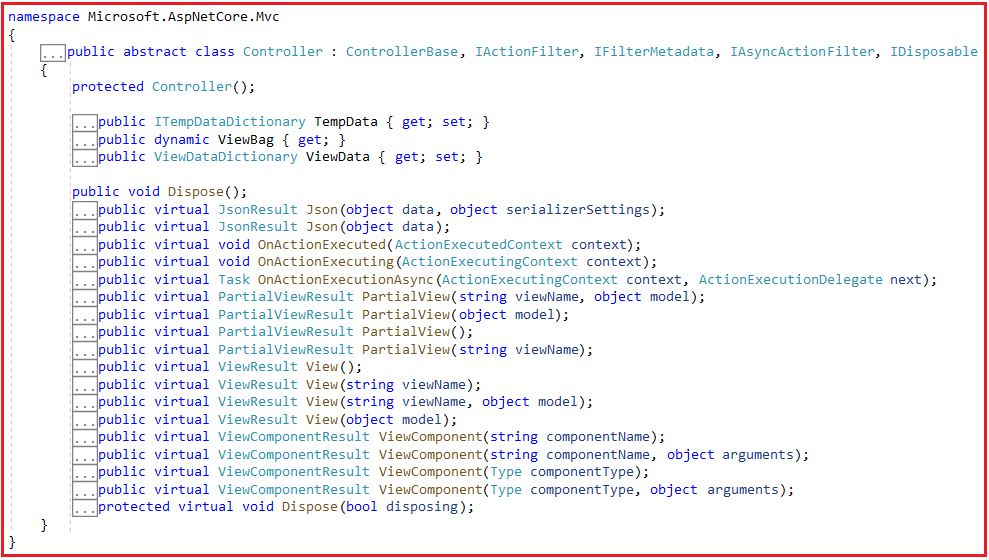


**Understanding StudentController:**

Now let us understand the StudentController class and the different components of this class. First of all, it is a class having a .cs extension. Open the StudnetController.cs class and you should get the following default code in it.

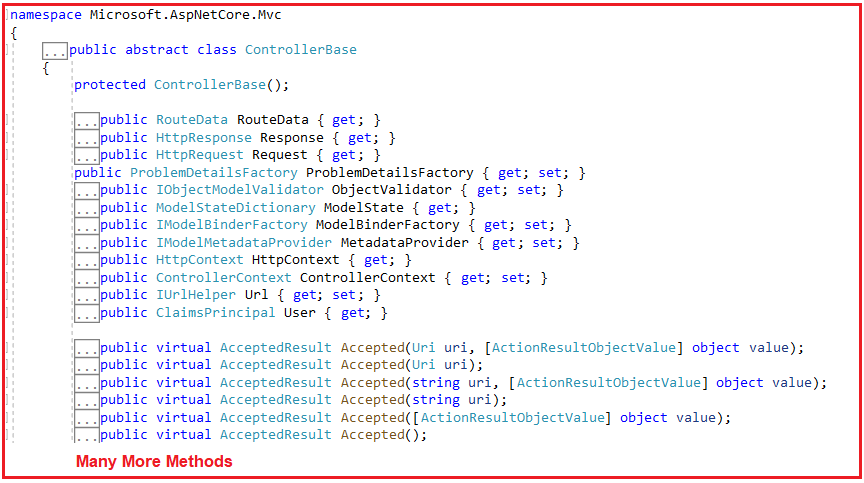


As you can see in the above image, the StudentController class is inherited from the Controller class, and this controller class present in **Microsoft.AspNetCore.Mvc** namespace and that’s why it import that namespace. Now right click on the Controller and select go to the definition and you will see the following definition of Controller class.



As you can see in the above image, the Controller has many methods (Json, View, PartialView, OnActionExecuting, etc) and properties (TempData, ViewBag, ViewData, etc.) and the point that you need to remember is these methods and properties are going to used when we are working with ASP.NET Core MVC Application. Again, if you look this Controller class is inherited from the ControllerBase class.

Let us see the ControllerBase class as well. Right-click on the ControllerBase class and select go to definition and you will see the following definition. Here, you will see the RequestData, Response, Request, ModelState, Routing, Model Binder, HttpContext, and many more which we are going to use as part of our ASP.NET Core MVC Application.



Now I hope you got clarity on ASP.NET Core MVC Controllers. Let us move forward and understand the next topic which is Action Methods.

**What are Action Methods?**

All the public methods of a controller class are known as Action Methods. Because they are created for a specific action or operation in the application. So, a controller class can have many related action methods. For example, adding a Student is an action. Modifying the student data is another action. Deleting a student is another action. So, the point that you need to remember is all the related actions should be created inside a particular controller.

An action method can return several types. Let us modify the HomeController as shown below where we have one method which returns all the student details. Intentionally we returned a string from this method but as we progress in this course, we will discuss the real implementation of this method. But for now, just the learning purpose we have to return a string.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** StudentController : Controller

**{**

**public** string GetAllStudents**()**

**{**

**return** "Return All Students";

**}**

**}**

**}**

**How to call an Action method of a Controller?**

When we get an HTTP Request on a Controller, it is actually the controller action method getting that call. So, whenever we saying we are hitting a controller, it means we are hitting an action method of a controller.

The default structure is:**http:domain.com/ControllerName/ActionMethodName**

As we are working with development using visual studio, the domain name is going to be our localhost with some available port number. So, if we want to access the GetAllStudents action method of the HomeController then the URL is something like below.

**http://localhost:<portnumber>/student/GetAllStudents**

Let us prove this. At this moment if you run the application and navigate to the above URL, then you will not get the output. This is because we have created this project using the Empty template. And the Empty Project template by default will not add the required MVC Service to the built-in dependency injection container as well as it will not set the required middleware to the application processing pipeline. So, let us set these two.

**Modifying the Startup class:**

Open the **Startup.cs** class file and then copy and paste the below code in it which will add the required MVC service to the dependency injection as well as will add the MVC middleware to the request processing pipeline.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreMVCWebApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Configuring the required service for MVC

services.AddMvc**()**;

//services.AddControllersWithViews();

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

//Configuring the MVC middleware to the request processing pipeline

endpoints.MapDefaultControllerRoute**()**;

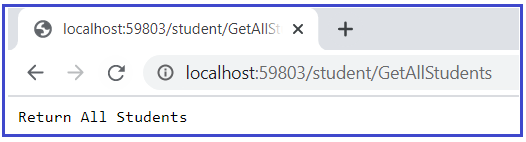
**})**;

**}**

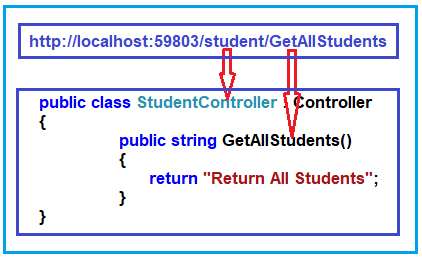
**}**

**}**

With the above change in place now run the application and navigate to the URL **http://localhost:<portnumber>/student/GetAllStudents** and you should get the output as expected as shown in the below image.



For a better understanding, how the above URL mapped to the GetAllStudents action method, please have a look at the following image.



**How to Pass Parameters in Action Methods?**

Let us understand this with an example. Now we want to search students based on the name. To do so, add the following action method inside the Student Controller.

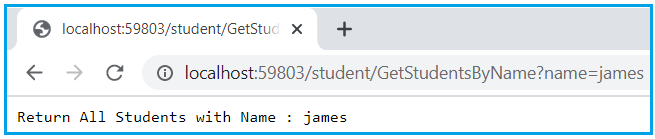
**public** string GetStudentsByName**(**string name**)**

**{**

**return** $"Return All Students with Name : {name}";

**}**

Now run the application and navigate to the URL **http://localhost:<portnumber>/student/GetStudentsByName?name=james** and see the output.



In this case, the query string parameter name is mapped with the GetStudentsByName action method name parameter. We will discuss the parameter mapping, the default mapping, and many more in our upcoming articles.

**When should we create a new controller?**

Whenever we need to define a new group of actions or operations into your applications, then you need to create a new controller. For example, to do operations of students you can create Student Controller. To manage the security of your application like login, logout, etc you can create a Security Controller.

**How many controllers can we have in a single application?**

It depends on the application. At least one controller is required to perform operations. Maximum n number of Controllers we can have into one application.

**Views in ASP.NET Core MVC**

**Views in ASP.NET Core MVC Application**

In this article, I am going to discuss **Views in the ASP.NET Core MVC** application. Please read our previous article before proceeding to this article where we discussed [**Controllers in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/controllers-asp-net-core-mvc/) application. As part of this article, we are going to discuss the following concepts related to Views in ASP.NET Core MVC Application.

1. **What are Views in ASP.NET Core MVC Application?**
2. **Roles of View.**
3. **Where View Files are Stored in ASP.NET Core MVC Application?**
4. **How to create View in ASP.NET Core?**
5. **How to return views from action methods?**
6. **What are the difference between View() and View(object model) Extension Methods?**
7. **How to specify the Absolute view file path?**
8. **What are the Advantages of Using Views in ASP.NET Core MVC Application?**

**What is a View in ASP.NET Core MVC Application?**

In the Model-View-Controller (MVC) pattern, the View is the component that contains logic to represent the model data (the model data provided to it by a controller) as a user interface with which the end-user can interact. The Views in MVC are HTML templates with embedded Razor mark-up which generate content that sends to the client. In our upcoming articles, we will discuss the Razor syntax in detail.

In ASP.NET Core MVC Application, a view is a file with “.cshtml” (for C# language) extension. The meaning of **cshtml = cs (c sharp) + html**. That means the view is the combination of programming language (ex C#) and HTML (Hypertext Mark-up Language).

The Views in ASP.NET Core MVC Application are generally returned from the Controller Action Method. We use the ViewResult return type to return a view from an action method.

**Role of View in ASP.NET Core MVC Application?**

A view in ASP.NET Core MVC Application is responsible for UI i.e. application data presentation. That means we display information about the website on the browser using the views only. A user generally performs all the actions on a view such as a button click, form, list, and other UI elements.

**Where are views placed in ASP.NET Core MVC Application?**

By default, Views are available inside the Views folder at the root. Usually, views are grouped into folder names with applications controller. Generally, views are grouped based on features.

Usually, each controller will have its own folder in which the controller-specific view files are going to be stored. The controller-specific folders are going to be created within the Views folder only. The point that you need to remember is the view file name is the same as the action method name of a controller with the “.cshtml” extension.

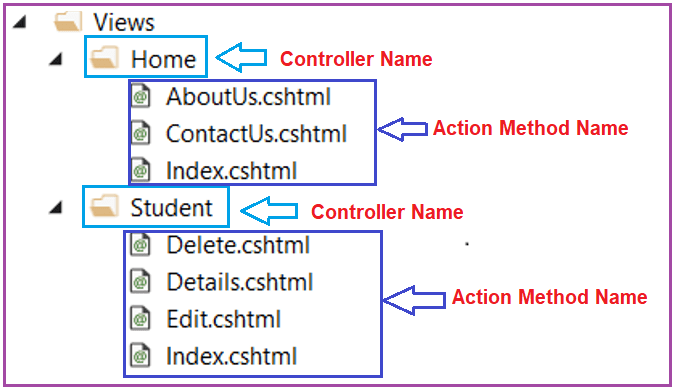
Let’s say, we have an ASP.NET Core MVC application with two controllers i.e. HomeController and StudentController. The HomeController is created with the following three action methods.

1. **AboutUs()**
2. **ContactUs()**
3. **Index()**

On the other hand, the StudentController is created with the following four action methods.

1. **Index()**
2. **Details()**
3. **Edit()**
4. **Delete()**

Then following is the folder and file structure of the Views



As you can see in the above image, a separate folder is created for each controller within the Views Folder. The Home Controller is represented by a Home folder and the Student Controller is represented by the Student folder inside the Views folder.

The Home folder contains the views for Index, AboutUs, and ContactUs webpages. So, whenever a user requests for any of these webpages then the Home Controller action method determines which of the above three views to use to build the webpage and return to the user.

Similarly, the Student folder contains the views for the Index, Details, Edit, and Delete webpages. So, whenever a user requests for any of these webpages then the Student Controller action method determines which of the above views to use in order to build the webpage and return to the user.

**Note:** In addition to action-specific views, we also have provided with partial views, layouts, and view components that can also be used to reduce the repetition and allow for reuse within the application’s views. We will discuss each of these in our upcoming articles.

**Understanding Views with an Example in ASP.NET Core MVC Application:**

As we are discussing everything from scratch, so let us first create an ASP.NET Core Web Application with the Empty Project template. By default, the Empty Project does not include Controllers and Views.

**Step1: Setting MVC Service and MVC Middleware**

In order to make your ASP.NET Core Web Application as an MVC application, we need to [**set up required MVC Services and MVC Middleware**](https://dotnettutorials.net/lesson/setup-mvc-asp-net-core-application/) to the request processing pipeline. And this can be done within the ConfigureServices and Configure method of the Startup class which is present inside the Startup.cs class file.

So, open the **Startup.cs** file and then copy and paste the below code in it which will add the required MVC services to the built-in dependency injection container as well as will add the MVC middleware to the request processing pipeline.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreMVCWebApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

//Configuring the required service for MVC

services.AddMvc**()**;

//services.AddControllersWithViews();

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

//Configuring the MVC middleware to the request processing pipeline

endpoints.MapDefaultControllerRoute**()**;

**})**;

**}**

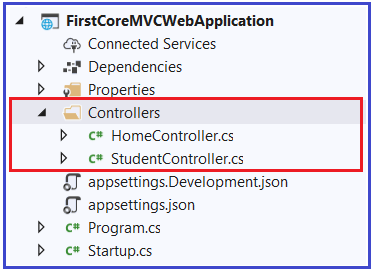
**}**

**}**

**Step2: Creating Controllers**

First, create a folder at the root level of the project with the name **Controllers** and then add two controllers within this folder with the name HomeController and StudentController. If you are new to Controllers then please read our previous article where we discussed Controllers in detail as well as we also discuss [**how to create a controller in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/controllers-asp-net-core-mvc/) Application. Here, while creating the Controllers you need to select the **MVC Controller – Empty** template.

At this time your project folder structure should be looks like below.



Open **HomeController** and then copy and paste the below code in it.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**()**;

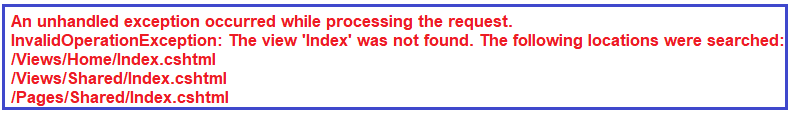
**}**

**}**

**}**

As you can see in the above HomeController code, we have only one action method i.e. Index. As the return type of the **Index()** action method is **ViewResult**, so this action method is going to return a **view**. In order to return a view, here we are using the **View()** extension method which is provided by **Microsoft.AspNetCore.Mvc.Controller** Base class.

Now run the application and navigate to the **“/Home/Index”** URL or to the default URL and you will see the following error.



**Let us understand why we got the above error.**

As we are returning a view from the Index action method of Home Controller, by default the ASP.NET Core MVC Framework will look for a file with the name **Index.cshtml** in the following three locations.

1. First, it will look for the “**Index.cshtml**” file within the “**/Views/Home**” folder as the action method belongs to Home Controller.
2. Then it will look in the “**/Views/Shared/**” folder
3. Finally, it will try to find out the “**Index.cshtml**” file in the “**/Pages/Shared/**” folder.

If the requested “**.cshtml**” file found in any of the above folders, then the View generates the HTML and sends the generated HTML back to the client who initially made the request. On the other hand, if the requested file is not found in any of the above locations, then we will get the above error.

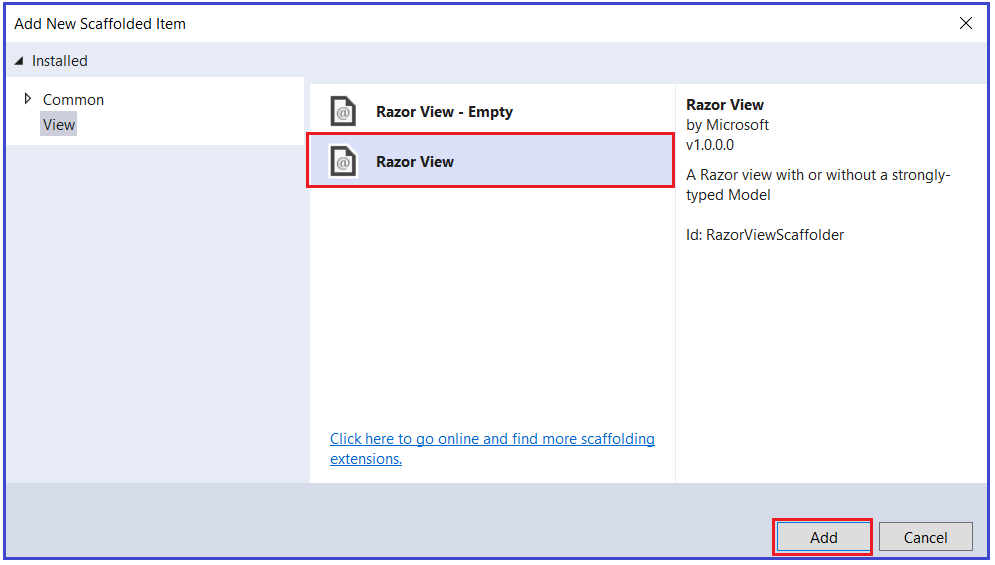
**Step3: Creating Index View**

As we already discussed views are going to be created in a special folder called Views. So, let us first add a folder to the project root level with the name Views. To do so, right-click on the Project and then select **Add => New Folder** from the context menu which will add a new folder and then rename the folder as **Views**.

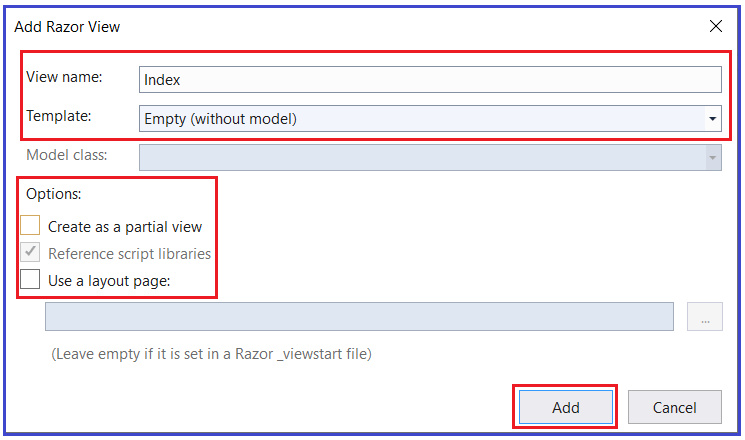
As we already discussed the Action Views are going to be created inside the folder whose name is the same as the Controller name. We want to create a view for the Index action method of Home Controller. So, let us first add the Home folder inside the View folder. To do so, right-click on the Views folder and then select the **Add => New Folde**r option which will add a new folder and then simply rename the folder as **Home**.

Once you created the Home folder, then we need to create a view within this Home folder. In ASP.NET Core MVC, we can add views in many different ways and we will discuss this one by one.

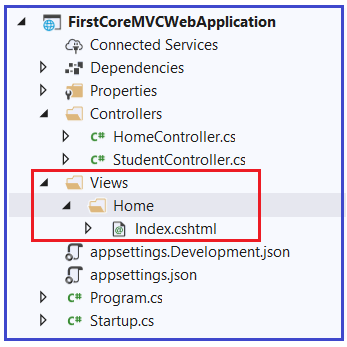
Right-click on the Home folder and then select **Add => View** option which will open the Add View window as shown in the below image. Here, you need to select the **Razor View**item and then click on the Add button.



Once you click on the Add button, then it will open the Add Razor View window. Here, you need to give the View name as Index, Select the Empty (without model) template, uncheck the create a partial view and use a layout page checkbox and then click on the Add button as shown in the below image.



Once you click on the Add button then it should create the Index.cshtml view within the Home folder as shown in the below image.



Now open **Index.cshtml** file and then copy and paste the following code in it.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Index**</title>**

**</head>**

**<body>**

**<h1>**Index view belongs to Views/Home folder**</h1>**

**</body>**

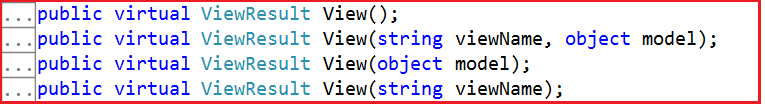
**</html>**

With the above changes in place, now run the application and you should get the output as expected as shown in the below image.



**Understanding View() Method in ASP.NET Core MVC Application:**

If you go to the definition of the **Controller** base class, then you will find there four overload versions of the View method as shown in the below image.

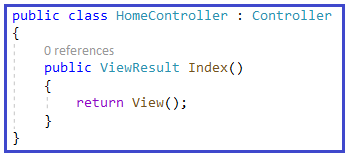


Let us understand the use and significance of each of the above-overloaded versions of the View Extension method.

**View() vs View(object model) Extension Methods:**

If you are using the **View()** or **View(object model)** extension method to return a view, then it will look for the view file with the same name as the action method name. If you want to pass some model data then you need to use the overloaded version which takes object model as input parameter else you can simply use the View() extension method which does not take any parameter.

For example, in the below code we are using the **View()** extension method which does not take any parameter to return a view from the Index action method of Home Controller. So, in this case, by default, the ASP.NET Core MVC framework will look for a view with the name **Index.cshtml** within the “**Views/Home**” folder.



**View(string viewName) vs** **View(string viewName, object model) Extension Methods:**

If you want to return a view from an action method which is name is different than the action method name then you need to use either View(string viewName) or View(string viewName, object model) Extension Methods. If you want to pass model data to the view then you need to use View(string viewName, object model) method else you can simply use the View(string viewName) method which takes the view name as a parameter,

To understand this concept, let’s create a view with the name **Test.cshtml** within the **Home** folder. Please follow the same step as we follow to create the index.cshtml view. Once you created the Test.cshtml view, then open the Test.cshtml file and copy-paste the below code in it.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Test**</title>**

**</head>**

**<body>**

**<h1>**Test view coming from Views/Home Folder**</h1>**

**</body>**

**</html>**

Now modify the Index action method of HomeController class as shown below to use the View extension method which takes the view name as a parameter.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**(**"Test"**)**;

**}**

**}**

**}**

Now run the application and navigate to the “**/Home/Index**” URL and you will see that the response is coming from the Test view as shown in the below image.



**Note**: You need to specify the view name without extension. Using this overloaded version, it is also possible to specify the file path. You can specify either the absolute path or relative path.

**How to specify the Absolute view file path?**

Let us modify the Index action method of the Home controller as shown below to specify the Absolute path of the view file. So here, the ASP.NET Core MVC framework will look for a view file with the name “**Test.cshtml**” within the “**Views/Home**” folder.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**(**"Views/Home/Test.cshtml"**)**;

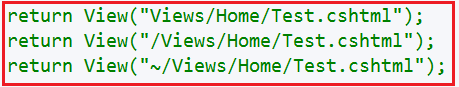
**}**

**}**

**}**

**Note**: When you are using the absolute path, then it is mandatory to use the **.cshtml** extension.

When you are using an absolute path, in order to get to the project’s root directory, you can use / or ~/. So. you can use any one of the following and all are going to do the same thing.



**Another way of Creating Views:**

Let us add a new action to the HomeController with the name About. So, please modify the HomeController as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**()**;

**}**

**public** ViewResult About**()**

**{**

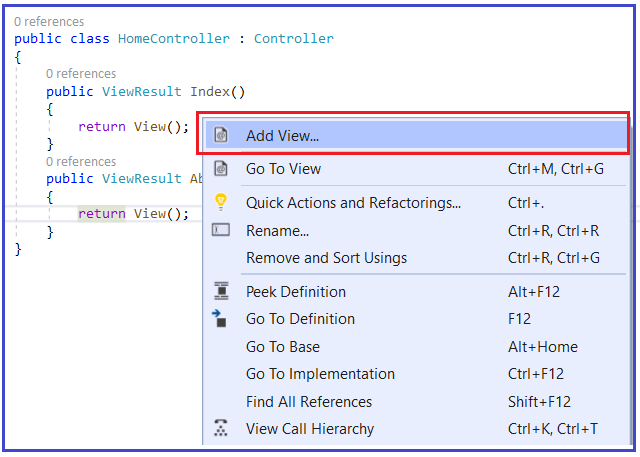
**return** View**()**;

**}**

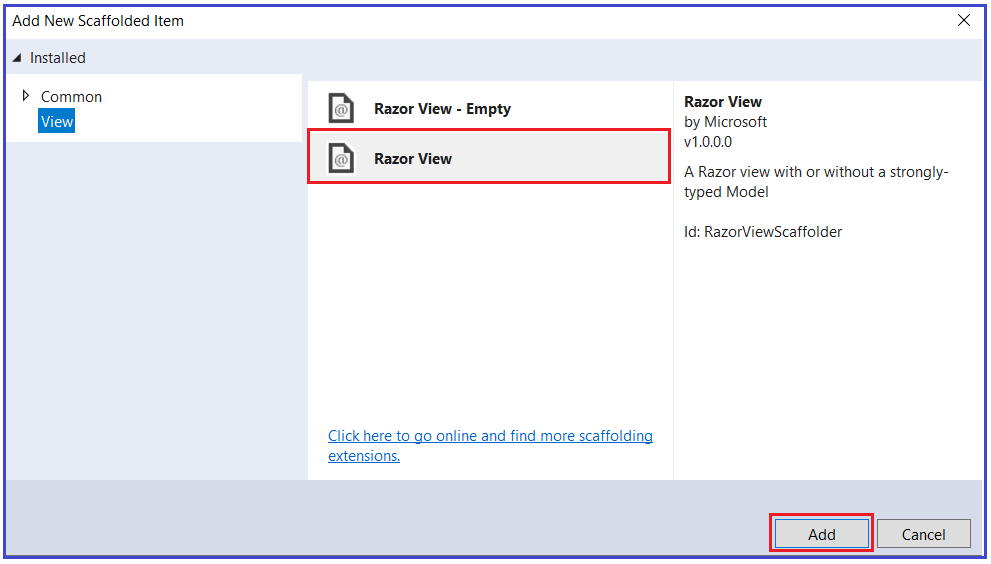
**}**

**}**

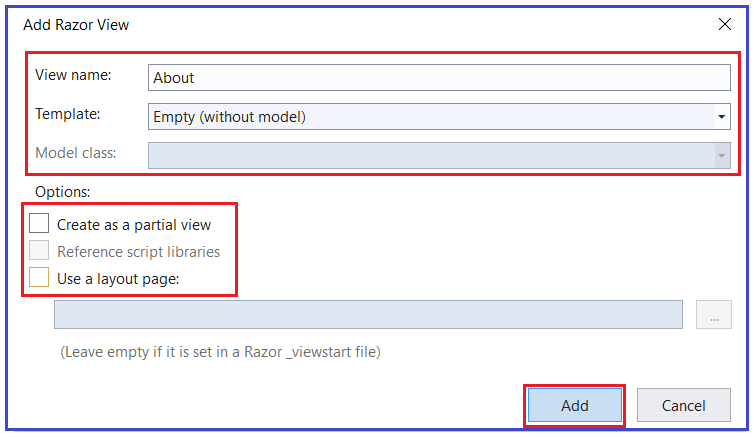
Now, right-click anywhere within the About action method and then click on the Add View from the context menu as shown in the below image.



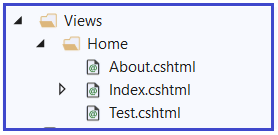
Once you click on the Add View then it will open Add New Scaffolded Item window as shown below where you need to select the Razor view and click on the Add button.



Once you click on the Add button, then it will open the Add Razor View window and by default, it will give the View name as About (action method name) and finally click on the Add button as shown in the below image.



Once you click on the Add button, then it will add the About.cshtml file within the View/Home folder as shown in the below image.



**What are the Advantages of Using Views in ASP.NET Core MVC Application?**

The Views in MVC application provides the separation of concerns (codes). It separates the user interface from the business logic or you can say from the rest of the application. The ASP.NET Core MVC views use the Razor syntax which makes it easy to switch between the HTML markup and C# code. The Common or repetitive aspects of the application’s user interface can be easily reused between views using layout and shared directives or partial views.

# Creating ASP.NET Core Application using MVC Template

## ****Creating ASP.NET Core Application using MVC Template****

In this article, I am going to discuss how to create**an ASP.NET Core application using MVC Project Template** (Web Application (Model-View-Controller)). Please read our previous article before proceeding to this article where we discussed [**Views in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/views-asp-net-core-mvc/) application.

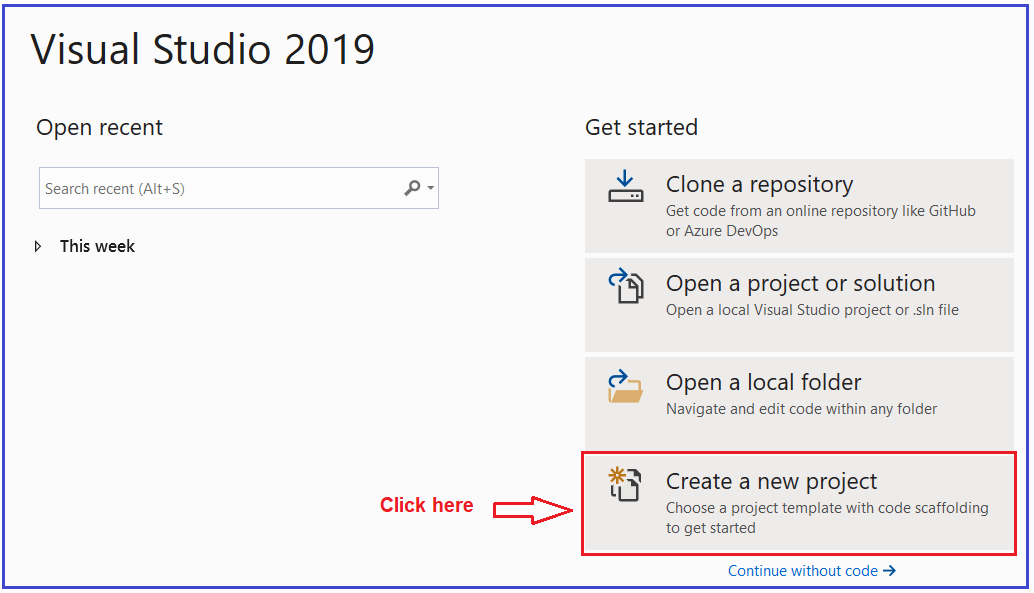
##### ****Creating an ASP.NET Core MVC Application:****

As of now, we have discussed everything using the Empty Project Template and understand the different parts of an ASP.NET Core Application. We also discussed how to set up the MVC Request pipeline in ASP.NET Core Web Application.

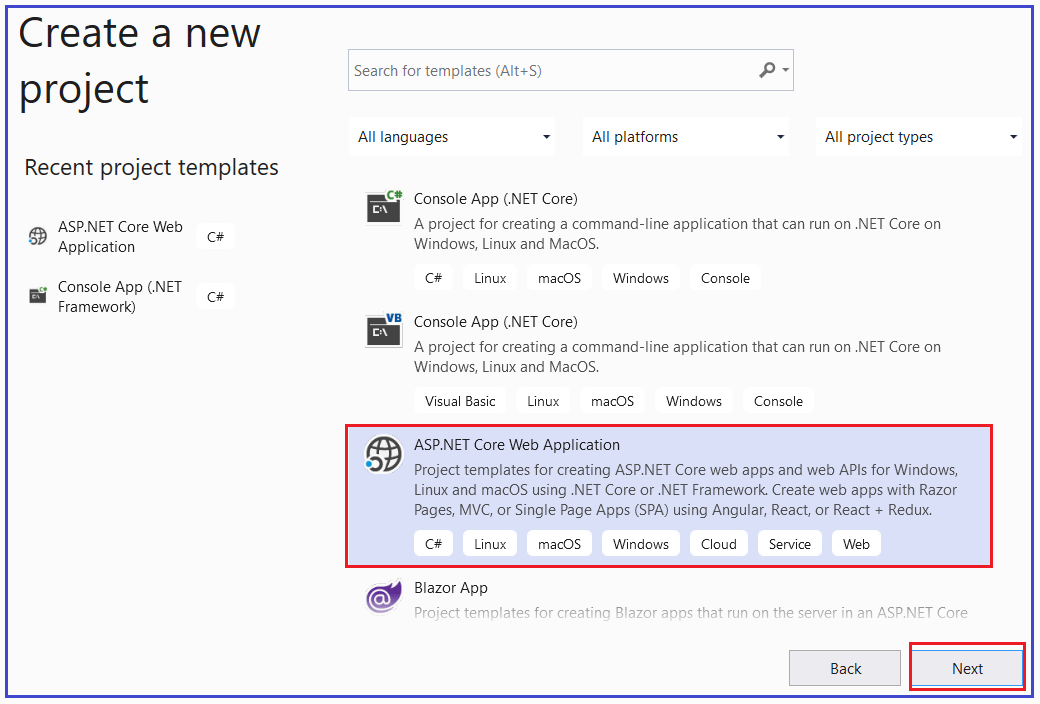
But the question is do we really need to set up everything manually to create an ASP.NET Core MVC Application? The answer is NO. The ASP.NET Core provides one built-in project template called MVC which will create an ASP.NET MVC Core MVC Application for us with the required setup. From this article onwards, in this ASP.NET Core MVC course, we are going to create the applications using the MVC Project template. Let us see the step by step procedure to create the same.

##### ****Creating an ASP.NET Core Application using MVC Project template:****

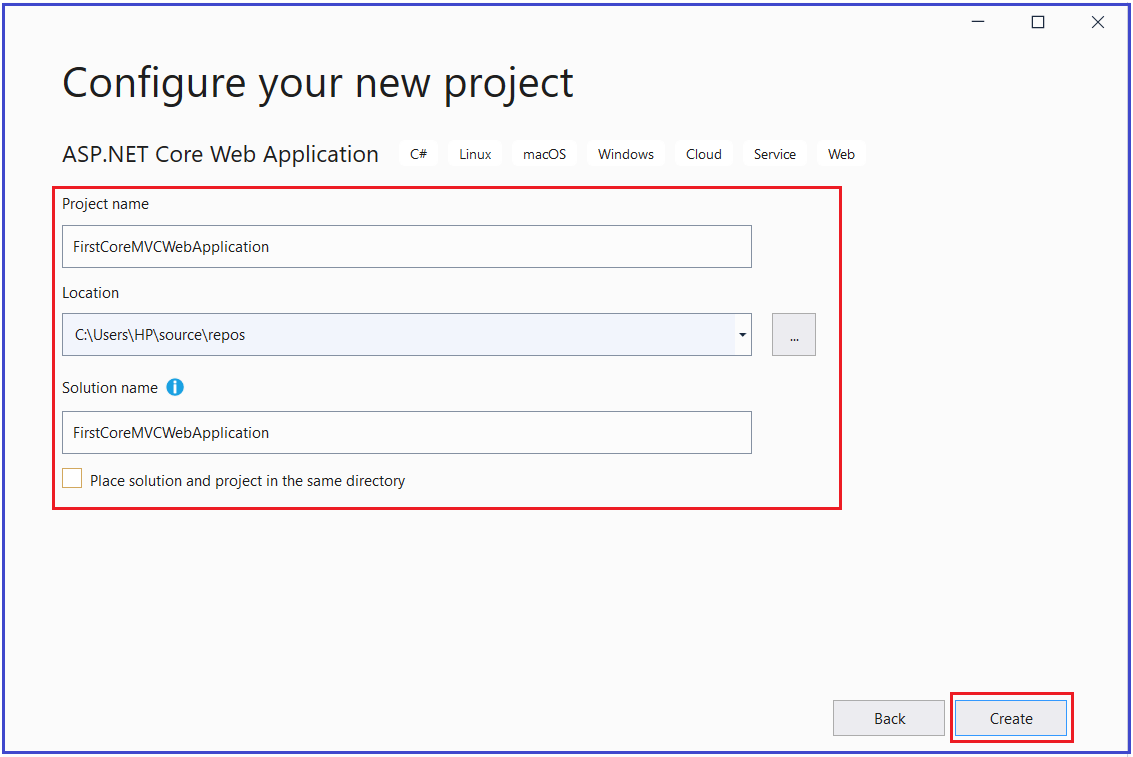
To create an ASP.NET Core Web Application with MVC Project template. First of all, open Visual Studio 2019 and then click on the Create a new project tab as shown in the below image.



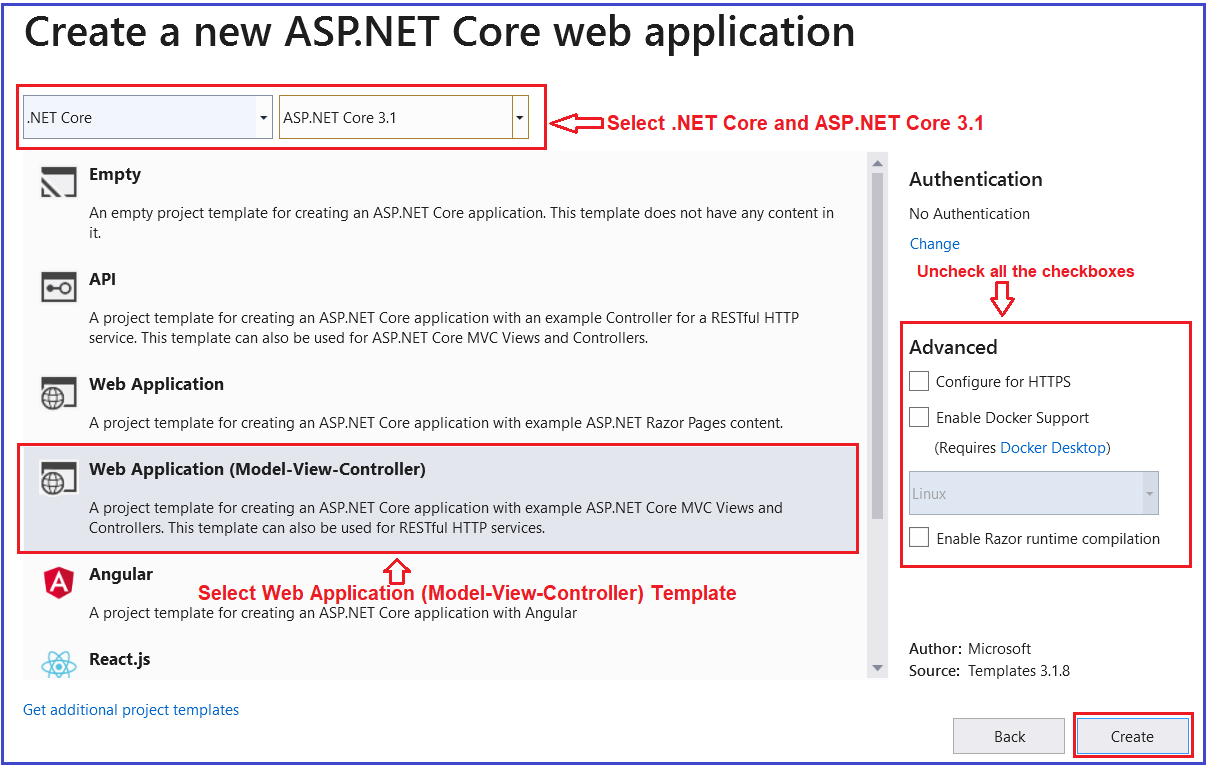
Once you click on the **Create a new project tab**, it will open the following Create a new project window. From this window, you need to select the **ASP.NET Core Web Application** template and then click on the **Next** button as shown in the below image.



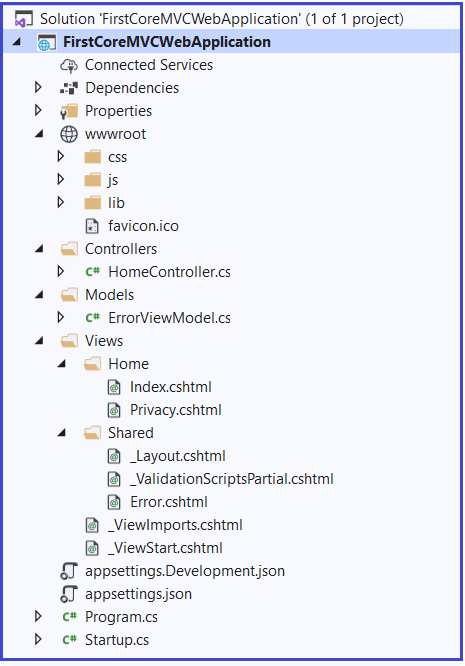
Once you click on the **Next** button, it will open the **Configure Your New Project** window. Here, you need to provide the necessary information to create a new ASP.NET Core project. First, give an appropriate name for your project (**FirstCoreMVCWebApplication**), set the location where you want to create this project, the solution name for the ASP.NET Core Web application. And finally, click on the **Create** button as shown in the image below.



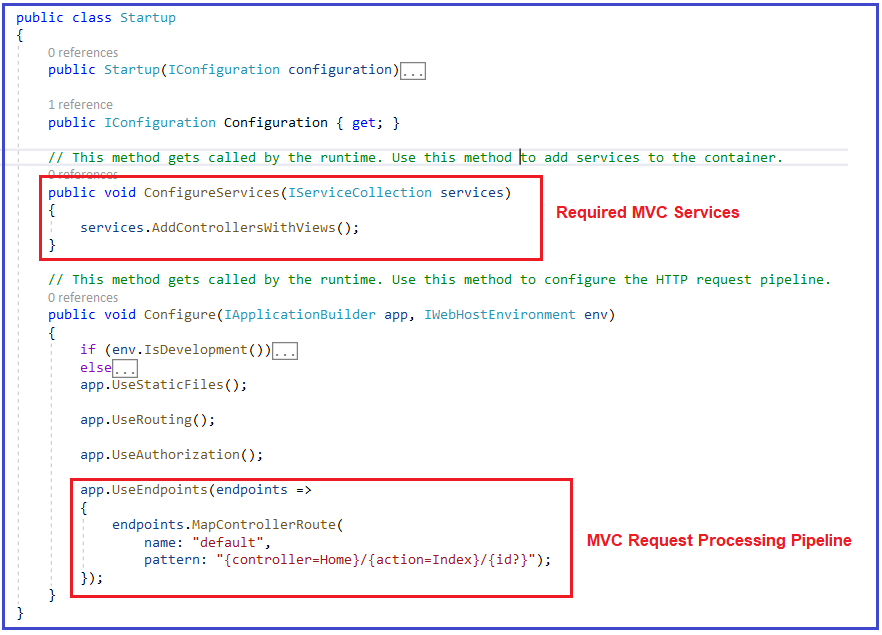
Once you click on the **create** button, it will open the following **Create a new ASP.NET Core Web Application** window where you need to select the project template i.e. which type of project you want to create. As we are going to create an MVC Application, so, select the **Web Application (Model-View-Controller)** Project template and uncheck all the checkboxes from the Advanced section and finally click on the **Create** button as shown in the below image. Make sure to select the .NET Core and ASP.NET Core 3.1 from their respectively drop-down list.



That’s it. Once you click on the **Create** Button, the project is going to be created with the Web Application (Model-View-Controller) i.e. MVC template with the following folder and file structure.

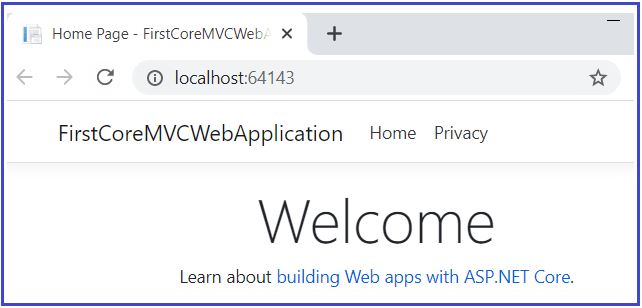


The MVC Project template by default includes the required setup for MVC. To confirm this, open the **Startup.cs** class file and you will see the required MVC Services and MVC Request processing pipeline are added by the framework as shown in the below image.



##### ****Running the Application:****

The MVC Project template creates the Home Controller with some views. Let us run the application and see the output as shown below.



**ViewData in ASP.NET Core MVC**

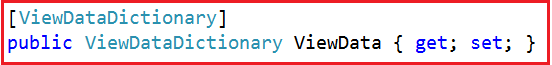
**ViewData in ASP.NET Core MVC Application**

In this article, I am going to discuss the use of **ViewData in ASP.NET Core MVC** application with examples. Please read our previous article before proceeding to this article where we discussed [**Views in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/views-asp-net-core-mvc/) application.

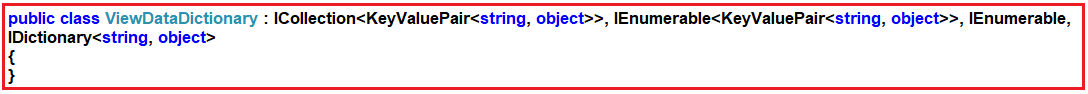
In ASP.NET Core MVC application, we can pass the data from a controller action method to a view in many different ways such as by using [**ViewBag**](https://dotnettutorials.net/lesson/viewbag-asp-net-core-mvc/), [**ViewData**](https://dotnettutorials.net/lesson/viewdata-asp-net-core-mvc/), and using a [**strongly typed model**](https://dotnettutorials.net/lesson/strongly-typed-view-asp-net-core-mvc/). In this article, I will show you how to use **ViewData** to pass the data from the controller action method to a view. The rest techniques i.e. [ViewBag](https://dotnettutorials.net/lesson/viewbag-asp-net-core-mvc/), and [strongly typed model](https://dotnettutorials.net/lesson/strongly-typed-view-asp-net-core-mvc/) are going to be discussed in our upcoming articles.

**What is ViewData in ASP.NET Core MVC Application?**

The ViewData in ASP.NET Core MVC is a dictionary of weakly typed objects which is derived from the ViewDataDictionary class.



Now, look at the definition of ViewDataDictionary as shown below.



As you can see in the above image, the ViewDataDictionary implements ICollection, IEnumerable, and IDictionary interfaces where it will store the data in the form of key-value pairs. Here, the key is of type String and the value is of type object i.e. we can store any type of data.

**What is the use of ViewData?**

1. ViewData is used to pass the data from the controller action method to a view and we can display this data on the view.
2. The ViewData is work on the principle of Key-value pairs. This type of binding is known as loosely binding.
3. We can pass any type of data in ViewData like normal integer, string, even though you can pass objects.
4. ViewData uses the ViewDataDictionary type.

**How to use ViewData?**

First, we need to create a new key in ViewData and then assign some data to it. The key should be in string format and you can give any name to it and then you can assign any data to this key.

**ViewData[“KeyName”] = “Some Data”;**

Since ViewData is a server-side code, hence to use it on view, we need to use the razor syntax i.e. @

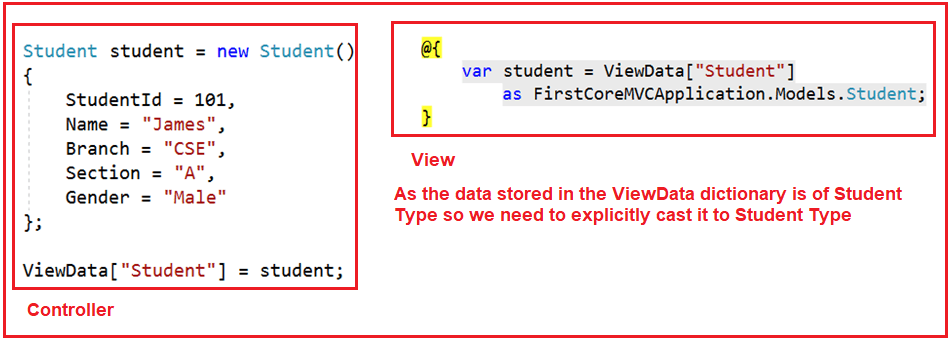
**@ViewData[“KeyName”]**

You can access the string data from the ViewData dictionary without casting the data to string type. But if you are accessing data other than the string type then you need to explicitly cast the data to the type you are expecting.

**Example: Accessing string data**



**Example: Accessing Student Data:**



**Note**: We can use n number of ViewData on a single view. ViewData can also be used to pass data from view to its layout view.

**ViewData Example in ASP.NET Core MVC Application:**

Let us see an example to understand how to use ViewData to pass data from a controller action method to a view. In our example, we want to pass three pieces of information to the view from the controller action method. One is the Title of the page, the second is the Header of the Page and the third one is the Student data that we want to display on the page.

First, create an ASP.NET Core Web Application with MVC (Model-View-Controller) project template. Once you create the project then add a class file with the name **Student.cs** in the Models folder. And then copy and paste the below code in it.

**namespace** *FirstCoreMVCWebApplication.Models*

**{**

**public** **class** Student

**{**

**public** string StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

**Modifying the HomeController:**

Now, modify the Home Controller class as shown below. Here, we are removing the existing code and adding one action method i.e. Details.

**using** *Microsoft.AspNetCore.Mvc;*

**using** *FirstCoreMVCWebApplication.Models;*

**namespace** *FirstCoreMVCWebApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Details**()**

**{**

//String string Data

ViewData**[**"Title"**]** = "Student Details Page";

ViewData**[**"Header"**]** = "Student Details";

Student student = new Student**()**

**{**

StudentId = "STD101",

Name = "James",

Branch = "CSE",

Section = "A",

Gender = "Male"

**}**;

//storing Student Data

ViewData**[**"Student"**]** = student;

**return** View**()**;

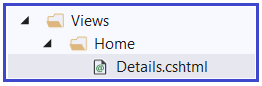
**}**

**}**

**}**

**Creating Details.cshtml view:**

In our previous article, we discussed the different ways to create Views in ASP.NET Core Application. Let us add a view with the name Details.cshtml within the Home Folder which is present inside the View Folder as shown below.



Now open Details.cshtml and then copy and paste the below code in it. As you can see in the below code, we directly access the string data from the ViewData without typecasting. But while accessing the Student data from the ViewData, we are typecasting it to the appropriate type.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewData["Title"]**</title>**

**</head>**

**<body>**

**<h1>**@ViewData["Header"]**</h1>**

@{

var student = ViewData["Student"]

as FirstCoreMVCWebApplication.Models.Student;

}

**<div>**

StudentId : @student.StudentId

**</div>**

**<div>**

Name : @student.Name

**</div>**

**<div>**

Branch : @student.Branch

**</div>**

**<div>**

Section : @student.Section

**</div>**

**<div>**

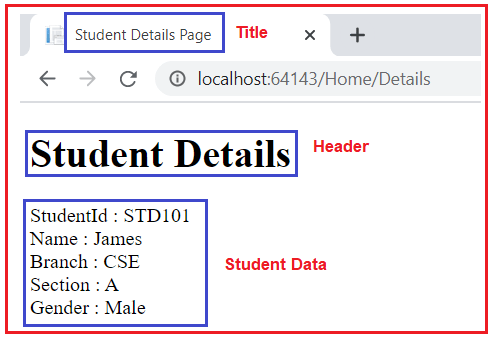
Gender : @student.Gender

**</div>**

**</body>**

**</html>**

Now run the application and navigate to the “**/Home/Details**” URL and you will see the data as expected as shown below.



**Points to Remember:**

The ViewData is dynamically resolved at runtime, as a result, it does not provide any compiles time error checking as well as we do not get any intelligence. For example, if we miss-spell the key names then we wouldn’t get any compile-time error. We get to know about the error only at runtime.

The ViewData only transfers the data from the controller action method to a view, but not vice-versa. That means it is valid only during the current request.

**ViewBag in ASP.NET Core MVC**

**ViewBag in ASP.NET Core MVC Application**

In this article, I am going to discuss the use of **ViewBag in ASP.NET Core MVC** application with examples. Please read our previous article before proceeding to this article where we discussed [**ViewData in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/viewdata-asp-net-core-mvc/) application. As we already discussed in the ASP.NET Core MVC application, we can pass the data from a controller to a view using ViewData, ViewBag, and strongly typed view model. As part of this article, we are going to discuss the following pointers.

1. **What is ViewBag in ASP.NET Core MVC?**
2. **How to Pass and Retrieve data From ViewBag in ASP.NET Core MVC?**
3. **Example to understand ViewBag in ASP.NET Core MVC**
4. **Difference between ViewData and ViewBag in ASP.NET Core MVC**

**What is ViewBag in ASP.NET Core MVC?**

The ViewBag in ASP.NET Core MVC is one of the mechanisms to pass the data from a controller action method to a view. If you go the Controller base class, then you will find the following signature of the ViewBag property.

ViewBag in ASP.NET Core MVC

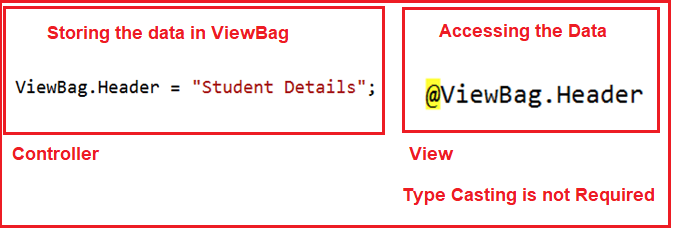
So the ViewBag is a dynamic property of the Controller base class. The dynamic type is introduced in C# 4.0. It is very much similar to the var keyword that means we can store any type of value in it but the type will be decided at run time rather than compile-time.

The ViewBag transfers the data from the controller action method to a view only, the reverse is not possible.

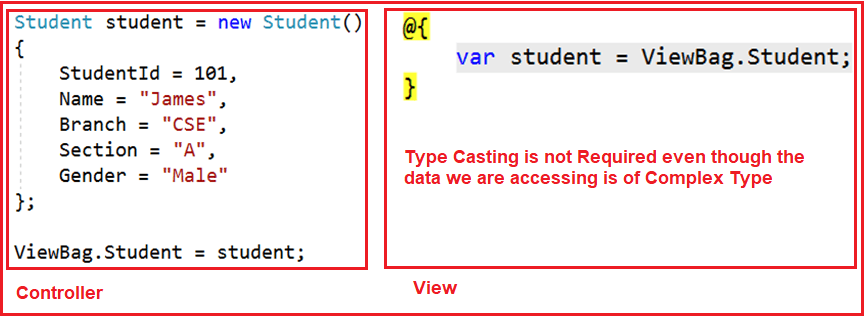
**How to Pass and Retrieve data From ViewBag in ASP.NET Core MVC?**

The point that you need to keep in mind is, ViewBag is operating on the dynamic data type. So we don’t require typecasting while accessing the data from a ViewBag. It does not matter whether the data that we are accessing is of type string or any complex type.

**ViewBag in ASP.NET Core MVC with String Type:**

****

**ViewBag in ASP.NET Core MVC with Complex Type:**

****

**Example of ViewBag in ASP.NET Core MVC:**

Let us see an example to understand how to use ViewBag to pass data from a controller to a view. We are going to work with the same example that we worked in our previous article with ViewData. So, modify the Details action method of HomeController class as shown below.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Details**()**

**{**

ViewBag.Title = "Student Details Page";

ViewBag.Header = "Student Details";

Student student = new Student**()**

**{**

StudentId = 101,

Name = "James",

Branch = "CSE",

Section = "A",

Gender = "Male"

**}**;

ViewBag.Student = student;

**return** View**()**;

**}**

**}**

**}**

As you can see in the above example, here we are using the dynamic properties Title, Header, and Student on the ViewBag.

**Accessing the ViewBag in a View in ASP.NET Core MVC**

Now we will see how to access the ViewBag data within an ASP.NET Core MVC view. So, modify the Details.cshtml view file as shown below.

**<html** xmlns="http://www.w3.org/1999/xhtml"**>**

**<head>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<h1>**@ViewBag.Header**</h1>**

@{

var student = ViewBag.Student;

}

**<div>**

StudentId : @student.StudentId

**</div>**

**<div>**

Name : @student.Name

**</div>**

**<div>**

Branch : @student.Branch

**</div>**

**<div>**

Section : @student.Section

**</div>**

**<div>**

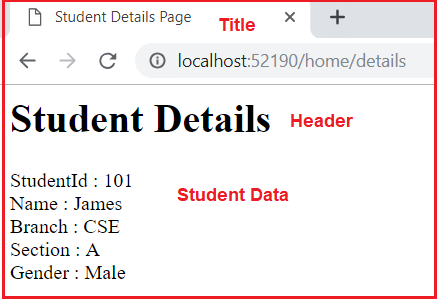
Gender : @student.Gender

**</div>**

**</body>**

**</html>**

As you can see, here we are accessing the data from the ViewBag using the same dynamic properties Title, Header, and Student.  Now run the application and navigate to the “**/Home/Details**” URL and you will see the data as expected on the webpage as shown in the below image.



The ViewBag is a dynamic property that is resolved at runtime; as a result, here also it will not provide compile-time error checking as well as intelligence support. For example, if we miss-spell the property names of the ViewBag, then we wouldn’t get any compile-time error rather we came to know about the error at runtime.

**Difference between ViewData and ViewBag in ASP.NET Core MVC**

1. In ASP.NET Core MVC, we can use both ViewData and ViewBag to pass the data from a Controller action method to a View.
2. The ViewData is a weakly typed dictionary object whereas the ViewBag is a dynamic property. Both ViewData and ViewBag are used to create a loosely typed view in MVC.
3. In ViewData, we use string keys to store and retrieve the data from the ViewData dictionary whereas in ViewBag we use the dynamic properties to store and retrieve data.
4. Both the ViewData keys and ViewBag dynamic properties are resolved only at runtime. As a result, both do not provide compile-time error checking and because of this, we will not get intelligence support.
5. So if we misspell the key names or dynamic property names then we will not get any compile-time error rather we came to know about the error only at run time. This is the reason why we rarely used ViewBag and ViewData in our application.

The best and preferred approach in MVC to pass data from a controller action method to a view is by using a strongly typed model object. When we use a strongly typed model object then only our view becomes a strongly typed view.

**Strongly Typed View in ASP.NET Core MVC**

**Strongly Typed View in ASP.NET Core MVC Application**

In this article, I am going to discuss how to create a **Strongly Typed View in ASP.NET Core MVC** application with examples. Please read our previous article before proceeding to this article where we discussed [**ViewBag in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/viewbag-asp-net-core-mvc/) application. As part of this article, we are going to discuss the following pointers.

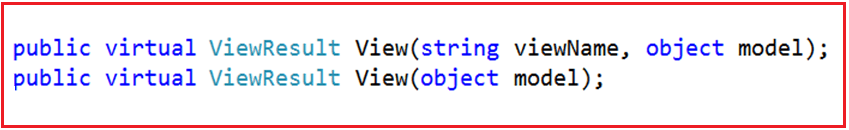
1. **Why we need a Strongly Typed View in ASP.NET Core MVC?**
2. **How to create a strongly typed view in ASP.NET Core?**
3. **What are the advantages of using a strongly typed view?**

**Why do we need Strongly Typed View in ASP.NET Core MVC?**

As we already discussed we can pass the model data to a view using many different ways such as ViewBag, ViewData, strongly typed model object, etc. When we passed the model data to a View using ViewBag or ViewData, then the view becomes a loosely typed view. In a loosely typed view, we will not get any intelligence as well as the compile-time error. With a strongly typed view, we will get both intelligence support as well as the compile-time error.

**Implementing Strongly Typed View in ASP.NET Core MVC**

In order to create a strongly-typed view, from the action method of the controller, we need to pass the model object as a parameter to the View() extension method. The Controller base class provides us the following two overloaded versions of View() extension method which we can use to pass the model object from the controller action method to a view.



Here we are going to use the overloaded version which takes only the model object as an input parameter. So, modify the Details action method as shown below to pass the student object as a parameter to the View extension method.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Details**()**

**{**

ViewBag.Title = "Student Details Page";

ViewBag.Header = "Student Details";

Student student = new Student**()**

**{**

StudentId = 101,

Name = "James",

Branch = "CSE",

Section = "A",

Gender = "Male"

**}**;

**return** View**(**student**)**;

**}**

**}**

**}**

**Changes in Details.cshtml View:**

In order to create a strongly-typed view in ASP.NET Core MVC, we need to specify the model type within the view by using the @model directive. As here, the Student class is going to be our model so we need to specify the model as shown below.

**@model FirstCoreMVCApplication.Models.Student**

The above statement will tell the view that we are going to use **FirstCoreMVCApplication.Models.Student** as the model for this view. The point that you need to remember is, here in the directive (**@model**), **m**is in lowercase and the statement should not be terminated with a semicolon.

Then in order to access the model object properties, you can simply use **@Model**, here the letter **M** is in uppercase. So, in our example, we can access the Student object properties such as Name, Gender, Branch, and Section by using **@Model.Name, @Model.Gender, @Model.Branch**, and **@Model.Section** respectively.

**Modify the Details.cshtml view file as shown below to make the view as strongly typed.**

@model FirstCoreMVCApplication.Models.Student

**<html** xmlns="http://www.w3.org/1999/xhtml"**>**

**<head>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<h1>**@ViewBag.Header**</h1>**

**<div>**

StudentId : @Model.StudentId

**</div>**

**<div>**

Name : @Model.Name

**</div>**

**<div>**

Branch : @Model.Branch

**</div>**

**<div>**

Section : @Model.Section

**</div>**

**<div>**

Gender : @Model.Gender

**</div>**

**</body>**

**</html>**

Now run the application and navigate to the “**/Home/Details**” URL and you will see the data as expected on the webpage.

**Advantages of using Strongly Typed View in ASP.NET Core MVC Application:**

We will get the following advantages when we use a strongly typed view in the ASP.NET Core MVC application.

1. It will provide compile-time error checking, as a result, we will get the intelligence support.
2. With intelligence support, the chances of mis-spelling the properties and making typographical errors are almost zero.
3. If we misspell the property name, then it comes to know at compile time rather than at runtime.

The best and preferred approach in ASP.NET Core MVC to pass data from a controller action method to a view is by using a strongly typed model object.

In our example, we are still using ViewBag to pass the Header and Title from the Controller action method to the View. Then definitely the question that comes to your mind is how we will pass the Header and Title to a strongly typed view. Well, in such scenarios we need to use a view specific model which is called View Model.

**ViewModel in ASP.NET Core MVC**

**ViewModel in ASP.NET Core MVC Application**

In this article, I am going to discuss **ViewModel in ASP.NET Core MVC** application with an example. Please read our previous article before proceeding to this article where we discussed [**Strongly Typed View in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/strongly-typed-view-asp-net-core-mvc/) application. As part of this article, we are going to discuss the following pointers.

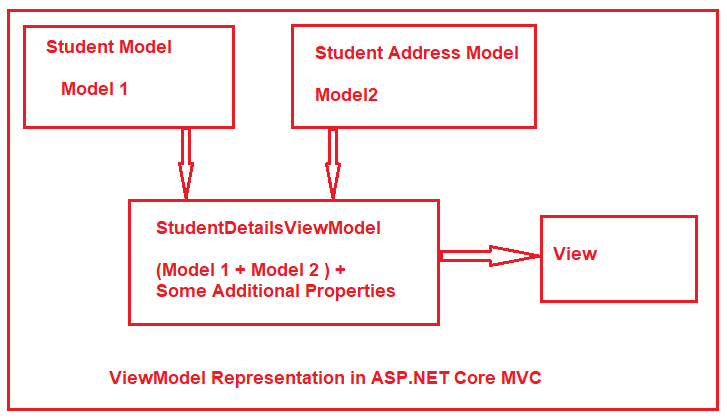
1. **What is a View Model in ASP.NET Core?**
2. **Why do we need the View Model?**
3. **How to implement the View Model in ASP.NET Core Application?**

**What is a ViewModel in ASP.NET Core MVC?**

In real-time applications, a single model object may not contain all the data required for a view. In such situations, we need to use ViewModel in the ASP.NET Core MVC application. So in simple words, we can say that a ViewModel in ASP.NET Core MVC is a model that contains more than one model data required for a particular view. Combining multiple model objects into a single view model object provides us better optimization.

**Understanding the ViewModel in ASP.NET Core MVC**:

The following diagram shows the visual representation of a view model in the ASP.NET Core MVC application.



Let say we want to display the student details in a view. We have two different models to represent the student data. The Student Model is used to represent the student basic details where the Address model is used to represent the address of the student. Along with the above two models, we also required some static information like page header and page title in the view. If this is our requirement then we need to create a view model let say StudentDetailsViewModel and that view model will contain both the models (Student and Address) as well as properties to store the page title and page header.

**Creating the Required Models:**

First, create a class file with the name **Student.cs** within the Models folder of your application. This is the model that is going to represent the basic information of a student such as a **name, branch, section**, etc. Once you create the **Student.cs** class file, then copy and paste the following code in it.

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

Next, we need to create the Address model which is going to represent the Student Address such as **City, State, Country,** etc. So, create a class file with the name Address.cs within the Models folder and then copy and paste the following code in it.

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Address

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** string State **{** **get**; **set**; **}**

**public** string Country **{** **get**; **set**; **}**

**public** string Pin **{** **get**; **set**; **}**

**}**

**}**

**Creating the View Model:**

Now we need to create the View Model which will store the required data that is required for a particular view. In our case its student’s Details view. This View Model is going to represent the Student Model + Student Address Model + Some additional data like page title and page header.

You can create the View Models anywhere in your application, but it is recommended to create all the View Models within a **folder** called **ViewModels** to keep the things organized.

So first create a folder at the root directory of your application with the name **ViewModels** and then create a class file with the name **StudentDetailsViewModel.cs** within the **ViewModels** folder. Once you create the **StudentDetailsViewModel.cs** class file, then copy and paste the following code in it.

**using** *FirstCoreMVCApplication.Models;*

**namespace** *FirstCoreMVCApplication.ViewModels*

**{**

**public** **class** StudentDetailsViewModel

**{**

**public** Student Student **{** **get**; **set**; **}**

**public** Address Address **{** **get**; **set**; **}**

**public** string Title **{** **get**; **set**; **}**

**public** string Header **{** **get**; **set**; **}**

**}**

**}**

We named the **ViewModel**class as **StudentDetailsViewModel**. Here the word **Student** represents the **Controller** name, the word **Details** represent the **action method name** within the Student Controller. As it is a view model so we prefixed the word **ViewModel**. Although it is not mandatory to follow this naming convention, I personally prefer it to follow this naming convention to organize view models.

**Creating Student Controller:**

Right-click on the Controllers folder and then add a new class file with the name StudentController.cs and then copy and paste the following code in it.

**using** *FirstCoreMVCApplication.Models;*

**using** *FirstCoreMVCApplication.ViewModels;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** StudentController : Controller

**{**

**public** ViewResult Details**()**

**{**

//Student Basic Details

Student student = new Student**()**

**{**

StudentId = 101,

Name = "Dillip",

Branch = "CSE",

Section = "A",

Gender = "Male"

**}**;

//Student Address

Address address = new Address**()**

**{**

StudentId = 101,

City = "Mumbai",

State = "Maharashtra",

Country = "India",

Pin = "400097"

**}**;

//Creating the View model

StudentDetailsViewModel studentDetailsViewModel = new StudentDetailsViewModel**()**

**{**

Student = student,

Address = address,

Title = "Student Details Page",

Header = "Student Details",

**}**;

//Pass the studentDetailsViewModel to the view

**return** View**(**studentDetailsViewModel**)**;

**}**

**}**

**}**

As you can see, now we are passing the view model as a parameter to the view. This is the view model that contains all the data required by the Details view. As you can notice, now we are not using any ViewData or ViewBag to pass the Page Title and Header to the view instead they are also part of the ViewModel which makes it a strongly typed view.

**Creating the Details View:**

First, add a folder with the name **Student** within the **Views** folder your project. Once you add the Student Folder, then you need to add a razor view file with the name **Details.cshtml** within the Student folder. Once you add the **Details.cshtml** view then copy and paste the following code in it.

@model FirstCoreMVCApplication.ViewModels.StudentDetailsViewModel

**<html** xmlns="http://www.w3.org/1999/xhtml"**>**

**<head>**

**<title>**@Model.Title**</title>**

**</head>**

**<body>**

**<h1>**@Model.Header**</h1>**

**<div>**

StudentId : @Model.Student.StudentId

**</div>**

**<div>**

Name : @Model.Student.Name

**</div>**

**<div>**

Branch : @Model.Student.Branch

**</div>**

**<div>**

Section : @Model.Student.Section

**</div>**

**<div>**

Gender : @Model.Student.Gender

**</div>**

**<h1>**Student Address**</h1>**

**<div>**

City : @Model.Address.City

**</div>**

**<div>**

State : @Model.Address.State

**</div>**

**<div>**

Country : @Model.Address.Country

**</div>**

**<div>**

Pin : @Model.Address.Pin

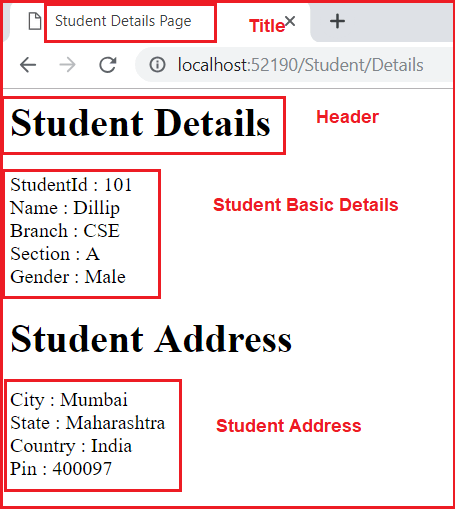
**</div>**

**</body>**

**</html>**

Now, the Details view has access to the **StudentDetailsViewModel** object that we passed from the controller action method using the View() extension method. By using the **@model** directive, we set **StudentDetailsViewModel** as the Model for the **Details** view. Then we access **Student, Address, Title,**and **Header** using **@Model**property.

Now run the application, and navigate to the “**/Student/Details**” URL and you will see the output as expected on the webpage as shown in the below image.



# Routing in ASP.NET Core MVC

## ****Routing in ASP.NET Core MVC Application****

In this article, I am going to discuss **Routing in ASP.NET Core MVC** application with examples. Please read our previous article where we discussed [**ViewModel in ASP.NET Core MVC Application**](https://dotnettutorials.net/lesson/view-model-asp-net-core-mvc/) with one real-time example. As part of this article, we are going to discuss the following pointers in detail.

1. **What is Routing in ASP.NET Core MVC?**
2. **What are the different types of Routing supported by ASP.NET Core MVC?**
3. **How Routing is working in ASP.NET Core?**
4. **Understanding Conventional based Routing.**

##### ****What is Routing in ASP.NET Core MVC?****

The Routing in ASP.NET Core MVC application is a mechanism in which it will inspect the incoming Requests (i.e. URLs) and then mapped that request to the controllers and their action methods. This mapping is done by the routing rules which are defined for the application. We can do this by adding the Routing Middleware to the request processing pipeline.

So, the ASP.NET Core Framework maps the incoming Requests i.e. URLs to the Controllers action methods based on the routes configured in your application. You can configure multiple routes for your application and for each route you can also set some specific configurations such as default values, constraints, message handlers, etc. If this is not clear at the moments then don’t worry we will discuss each and everything with examples.

##### ****What are the different types of Routing supported by ASP.NET Core MVC?****

In ASP.NET Core MVC application, you can define routes in two ways. They are as follows:

1. **Convention Based Routing**
2. **Attribute-Based Routing.**

##### ****What is Conventional Based Routing in ASP.NET Core MVC Application?****

In Conventional Based Routing, the route is determined based on the conventions defined in the route templates which will map the incoming Requests (i.e. URLs) to controllers and their action methods. In ASP.NET Core MVC application, the Convention based Routes are defined within the Configure method of the Startup.cs class file.

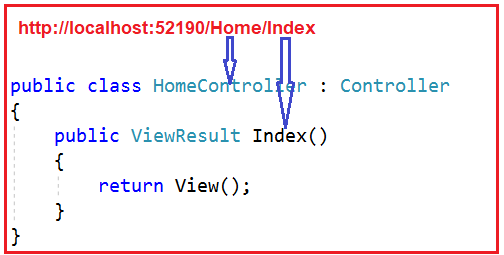
##### ****What is Attribute-Based Routing in ASP.NET Core MVC Application?****

In Attribute-Based Routing, the route is determined based on the attributes which are configured either at the controller level or at the action method level. We can use both Conventional Based Routing and Attribute-Based Routing in a single application.

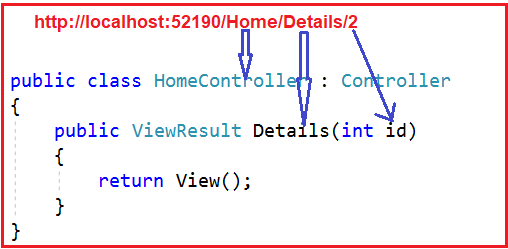
In this article, we are going to discuss the Conventional Based Routing and in our upcoming article, we will discuss the Attribute-Based Routing.

##### ****Understanding Conventional Based Routing in ASP.NET Core MVC:****

In ASP.NET Core MVC application, it is the controller action method that is going to handle the incoming Requests i.e. URLs. For example, if we issue a request to the **“/Home/Index**” URL, then it is the Index action method of Home Controller class which is going to handle the request as shown in the below image.



Similarly, if you issue a request to the **“/Home/Details/2**” URL, then it is the Details action method of the Home Controller class which is going to process that request as shown in the below image. Here the parameter value 2 is automatically mapped to the id parameter of the Details action method.

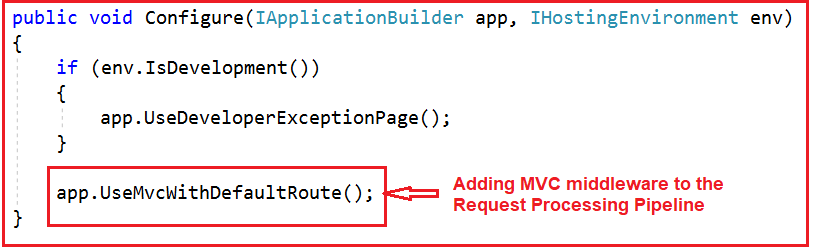


Now, the question that should come to your mind is, we have not explicitly defined any routing rules for the application, then how does this mapping is done i.e. how the “**/Home/Index**” URL is mapped to the Index action method and how “**/Home/Details/2**” URL is mapped to the Details action method of the Home Controller class.

This is actually done by the MVC Middleware which we registered in the application’s request processing pipeline.

##### ****Understanding the Default Route in ASP.NET Core MVC Application:****

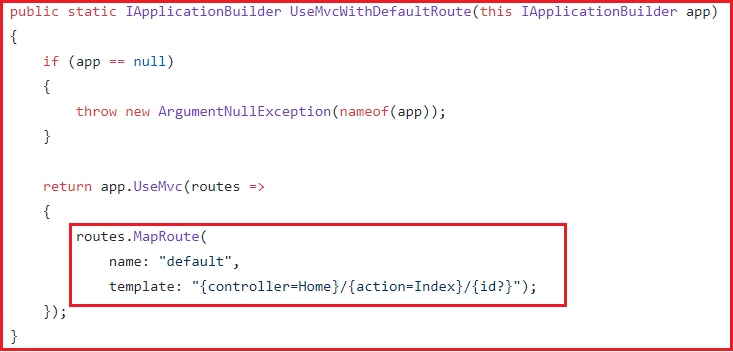
As we already discussed in our previous article that we can add the required MVC middleware into the request processing pipeline either by calling the **UseMvcWithDefaultRoute()** method or by calling the **UseMvc()** method within in the Configure() method of the **Startup.cs** class file as shown in the below image. As of now, we are using the **UseMvcWithDefaultRoute()** middleware.



Let us have a look at the implementation of the **UseMvcWithDefaultRoute()** method by visiting the following GitHub URL.

<https://github.com/aspnet/Mvc/blob/release/2.2/src/Microsoft.AspNetCore.Mvc.Core/Builder/MvcApplicationBuilderExtensions.cs>

**The implementation is given as shown below.**



As you can see in the above implementation, this method internally calls the **UseMvc()** method which will add the default route into the application’s request processing pipeline. The default route is created with the following URL template:

**{controller=Home}/{action=Index}/{id?}**

##### ****Understanding The Route Template:****

The above default route template maps most URLs that have the following pattern.

**http://localhost:52190/Student/Details/2**

The first segment path of the URL i.e. “**/Student**” is mapped to the “**StudentController**“. As you can see in the URL we do not have the word Controller with the first segment path of the URL. But it maps to the StudentController, this is because when ASP.NET Core MVC Framework finds the word /Student as the first segment path of URL, then it internally appends the word Controller.

The second segment path of the URL i.e. “**/Details**” is mapped to the “**Details(int id)**” action method of the HomeController class and the third segment path of the URL i.e. “**2**” is mapped to the “**id**” parameter of the Details(int id) action method.

As you can see in the default route template **“{controller=Home}/{action=Index}/{id?}“**, we have a question mark at the end of the id parameter which makes the parameter id as optional. That means the following two requests now map to the same Details action method of the Home Controller class.

**/Home/Details/1**  
**/Home/Details**

In the default route template **“{controller=Home}/{action=Index}/{id?}“,**the value “**Home**” in **{controller=Home}** is the default value for the Controller. Similarly the value “**Index**” in **{action=Index}** is the default value for the action method.

That means if we navigate to the applications root URL then as shown, then that request is going to be handled by the Index action method of the Home Controller class.

###### **http://localhost:52190**

The following two URLs are also mapped to the Index action method of the HomeController class.

**http://localhost:52190/Home**

**http://localhost:52190/Home/Index**

For most of the ASP.NET Core MVC applications, the default route works fine. For example, create a controller with the name as StudentController and then copy and paste the following code in it.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** StudentController : Controller

**{**

**public** string Index**()**

**{**

**return** "Index() Action Method of StudentController";

**}**

**public** string Details**(**string id**)**

**{**

**return** "Details() Action Method of StudentController";

**}**

**}**

**}**

Now, the URL **“/student/index**” is mapped to the **Index()** action method of the **StudentController**class and the URL **“/student/details**” is mapped to the **Details()** action method of the **StudentController**.

**Custom Routing in ASP.NET Core MVC**

**Custom Routing in ASP.NET Core MVC Application**

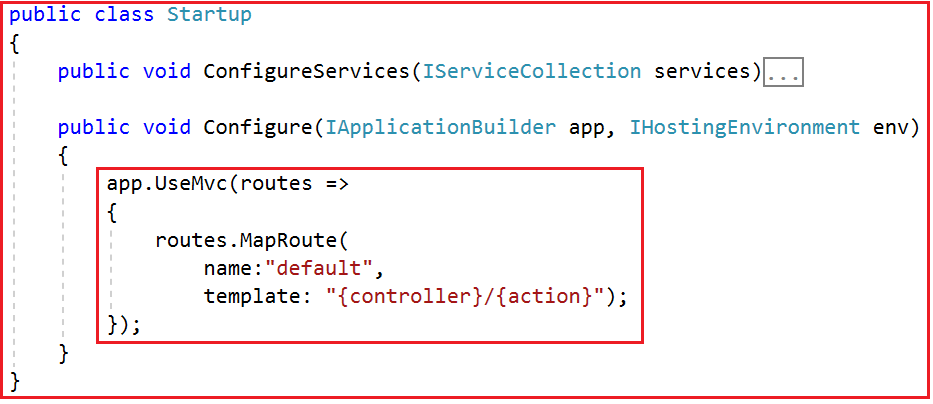
In this article, I am going to discuss **Custom Routing in the ASP.NET Core MVC** application. Please read our previous article before proceeding to this article where we discussed the basics of Routing as well as we also discussed the fundamental of [**Conventional based routing in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/routing-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers in detail.

1. **Creating Custom Routing in ASP.NET Core MVC Application.**
2. **Understanding Route Constraints?**
3. **How to define Optional Parameters in Route?**
4. **Providing Default Route Values.**

**Note:** We are going to work with the same example that we created in our previous article.

**Custom Routing in ASP.NET Core MVC Application:**

If you want to define your own route then you need to the **UseMvc** middleware instead of **UseMvcWithDefaultRoute()**. Within the Configure method of the Startup class file use the **UseMVC** middleware and within that middleware make a call to the **MapRoute** method to define your own route as shown in the below image.



The above example is the simplest possible convention-based route for an ASP.NET Core MVC application. Now run the application and navigates to the following URLs and you will see the output as expected.

**http://localhost:52190/Student/Details**

**http://localhost:52190/Student/Index**

This is working fine. However, what if we wanted to have more specific routes? Say, something like:

**http://localhost:52190/Student/Details/20**

**http://localhost:52190/Student/Index/10**

If you want your controller action methods to match the above URLs, then you need to use a feature called Route Constraints in ASP.NET Core MVC Application.

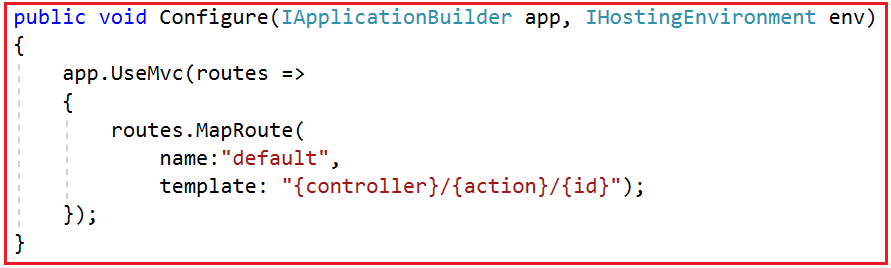
**Route Constraints in ASP.NET Core MVC Application:**

Let’s say we want to create a route that will match the following URL.

**http://localhost:52190/Student/Index/10**

**http://localhost:52190/Student/Details/20**

In order to achieve this, one of the simplest ways is to define a route as shown below:



Now modify the StudentController as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** StudentController : Controller

**{**

**public** string Index**(**string count**)**

**{**

**return** "Index() Action Method of StudentController";

**}**

**public** string Details**(**string id**)**

**{**

**return** "Details() Action Method of StudentController";

**}**

**}**

**}**

Now run the application and navigate to the respective URLs and you will see that methods are executed as expected.

The problem with the above route is that it can accept any type of values. Here instead of an integer, if you pass string values then also it accepts and executes the action methods as shown below.

**http://localhost:52190/Student/Details/ABC**

If you want to restrict the id parameter value to be an integer only, then you need to use a concept called route constraint as shown below:

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *FirstCoreMVCApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

app.UseMvc**(**routes =**>**

**{**

routes.MapRoute**(**

name:"default",

template: "{controller}/{action}/{id:int}"**)**;

**})**;

**}**

**}**

**}**

**Note:** The **{id:int}** in the template section specifies that whatever is in this part of the URL must be an integer, otherwise the URL does not map to this route.

With the above changes, now run the application and navigate to the following URL and you will see a 404 error. This is because here we are passing the Id parameter value as ABC.

**http://localhost:52190/Student/Details/ABC**

**http://localhost:52190/Student/index/ABC**

Now pass the id parameter value as an integer and you should get the output as expected. There are many route constraints are available that you can use. Some of them are as follows.

1. Int
2. Bool
3. Datetime
4. Decimal
5. Guid
6. length(min,max)
7. alpha
8. range(min,max)

For the list of all available route constraints, please find the following MSDN Article.

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.2#route-constraint-reference>

**Optional Parameters:**

Before understanding the Optional Parameters, let us first change the StudentController as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** StudentController : Controller

**{**

**public** string Index**()**

**{**

**return** "Index() Action Method of StudentController";

**}**

**public** string Details**(**string id**)**

**{**

**return** "Details() Action Method of StudentController";

**}**

**}**

**}**

As you can see, the Index action method does not take any parameter while the Details action method takes one parameter. Now we need to invoke the Index action method without parameter. On the other hand, we need to make the id parameter of the Details action method as optional. It means the Details action method should be invoked using the following two URLs.

**http://localhost:52190/Student/Details**

**http://localhost:52190/Student/Details/10**

In order to achieve this, we need to use optional parameters in our convention-based routes by adding a question mark “**?**” to the optional parameter’s constraint as shown below.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *FirstCoreMVCApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

app.UseMvc**(**routes =**>**

**{**

routes.MapRoute**(**

name:"default",

template: "{controller}/{action}/{id:int?}"**)**;

**})**;

**}**

**}**

**}**

**Note:** You can define only one optional parameter per route, and that optional parameter must be the last parameter.

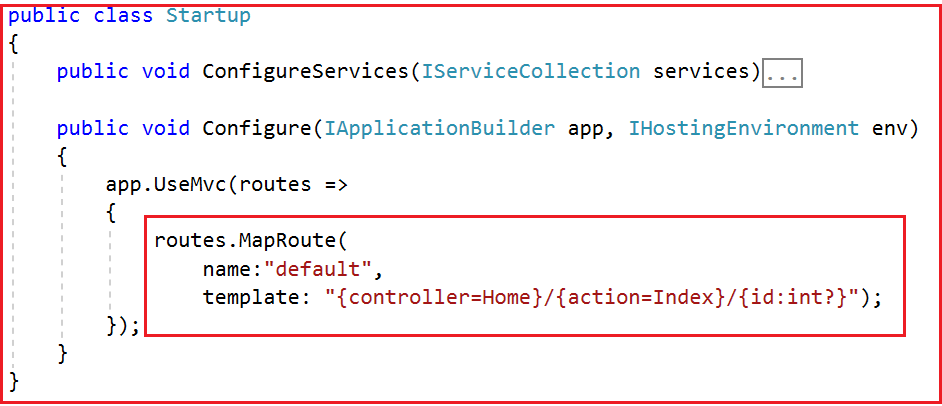
**Providing Default Route Values in ASP.NET Core MVC Application:**

Using Default values we can specify what happens if parts of the route are not provided in the URL. For example, when we navigate to the following two URLs

**http://localhost:52190/**

**http://localhost:52190/Home**

We want to map the above two URLs to the Home Controller and Index action method of the Application. In order to achieve this, we need to provide default route values while defining the routes as shown below.



**Modify the Home Controller as shown below.**

**public** **class** HomeController : Controller

**{**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**}**

Now run the application and navigate to the following URLs and you will see the output as expected.

**http://localhost:52190/**

**http://localhost:52190/Home**

You can also map the default values by using the defaults property as shown below.

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

app.UseMvc**(**routes =**>**

**{**

routes.MapRoute**(**

name: "default",

template: "{controller}/{action}/{id:int?}",

defaults: new **{** controller = "Home", action = "Index" **})**;

**})**;

**}**

**}**

**Attribute Routing in ASP.NET Core MVC**

**Attribute Routing in ASP.NET Core MVC Application**

In this article, I am going to discuss **Attribute Routing in ASP.NET Core MVC Application** with examples. Please read our previous article before proceeding to this article where we discussed the [**Conventional Based Custom Routing in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/custom-routing-inasp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers.

1. **Need and Use of Attribute Routing in ASP.NET Core MVC Application.**
2. **What is Attribute Routing in ASP.NET Core MVC?**
3. **Attribute Routing with Parameters in ASP.NET Core MVC Application**
4. **Attribute Routing with Optional Parameters in ASP.NET Core MVC Application**
5. **Controller and Action Method Names in Attribute Routing.**
6. **Attribute Routes at Controller Level**
7. **How to ignore the Route Template placed at the Controller Level?**

**Need and Use of Attribute Routing in ASP.NET Core MVC Application**

Before understanding the **Attribute Routing in ASP.NET Core MVC** Application, let us first do some changes to our application. First, modify the Configure() method of the Startup.cs file as shown below. If you notice in the code, here we are using the **UseMvc()**method without passing the default route template as a parameter.

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

app.UseMvc**()**;

**}**

**}**

With the above changes in the Configure method, now our application does not have any configured routes to handle the request and response.

**Now Modify the Home Controller as shown below.**

**public** **class** HomeController : Controller

**{**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**}**

Now, when you navigate to any of the following URLs, you will get the 404 errors. This is because at the moment we don’t have any configured route in our application to handle the request.

1. **http://localhost:52190**
2. **http://localhost:52190/home**
3. **http://localhost:52190/home/index**

Let us see how to use Attribute Routing to map the incoming URLs to the Index action method of Home Controller.

**Attribute Routing in ASP.NET Core MVC:**

With the help of ASP.NET Core Attribute Routing, you can use the Route attribute to define routes for your application. You can use the Route attribute either at the Controller level or at the Controller Action Methods level. When you apply the Route attribute at the Controller level, then it is applicable for all the action methods of that controller.

Let us modify the Home Controller as shown below. Here we have applied the Route Attribute at the Action method.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"Home"**)]**

**[**Route**(**"Home/Index"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**}**

**}**

If you notice, here we applied the Route() attribute 3 times on the Index() action method of Home Controller. The point that you need to remember is, with each instance of the Route attribute we specified a different route template. With the above three Route attribute, now we can access the Index() action method of the HomeController using the following 3 URLs.

1. **http://localhost:52190**
2. **http://localhost:52190/home**
3. **http://localhost:52190/home/index**

Now run the application and navigate to the above three URLs and you will see the output as expected.

**Attribute Routing with Parameters in ASP.NET Core MVC Application:**

As we already discussed, with conventional based routing, we can specify the route parameters as part of the route template. We can also do the same with attribute routing. That means we can also define Route Attribute with parameters. To understand this, modify the Home Controller as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"Home"**)]**

**[**Route**(**"Home/Index"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**[**Route**(**"Home/Details/{id}"**)]**

**public** string Details**(int** id**)**

**{**

**return** "Details() Action Method of HomeController, ID Value = " + id;

**}**

**}**

**}**

As you can see in the above code, the Details() action method has the id parameter. Notice in the route template, we also specified the id parameter. So the URL (/Home/Details/10) will execute the Details(int id) action method and maps the value “10” to the “id” parameter of the Details action method. This is done by a process called **Model binding**which will discuss in our upcoming articles. Now, run the application and navigate to the following URL and you will see the output as expected.

**http://localhost:52190/Home/Details/10**

**Attribute Routing with Optional Parameters in ASP.NET Core MVC Application:**

Like conventional based routing, we can also make a parameter as optional in Attribute Routing. To make the Route parameter optional, simply add a question mark “**?**” at the end of the parameter.

In our example, at the moment, the Details(int id) action method of the HomeController is executed only if we pass the id parameter value. If we have not passed the id parameter value in the URL, then we will get 404. For example, at the moment if we navigate to the following URL we will get a 404 error.

**http://localhost:52190/Home/Details**  
Let us modify the Route attribute of the Details action method as shown below to make the route parameter **“id”** as optional by adding a **“?”** at the end.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"Home"**)]**

**[**Route**(**"Home/Index"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**[**Route**(**"Home/Details/{id?}"**)]**

**public** string Details**(int** id**)**

**{**

**return** "Details() Action Method of HomeController, ID Value = " + id;

**}**

**}**

**}**

Now run the application and navigate to the following URL and you will see the output as expected instead of 404 error.

**http://localhost:52190/Home/Details/**

**Controller and Action Method Names in Attribute Routing:**

With attribute routing in ASP.NET Core MVC Application, the controller name and action method names do not play any role. To understand this, modify the Home Controller as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"MyHome"**)]**

**[**Route**(**"MyHome/Index"**)]**

**public** string StartPage**()**

**{**

**return** "StartPage() Action Method of HomeController";

**}**

**}**

**}**

As you can see, we have specified the Route attribute three times StartPage() action method of the HomeController. So, this StartPage action method is going to be executed for the following 3 URLs.

**/**  
**/MyHome**  
**/MyHome/Index**

**Attribute Routes at Controller Level:**

In the ASP.NET Core MVC application, it is also possible to apply the Route() attribute on the Controller class as well as on individual action methods. If you want to make the attribute routing less repetitive, then you need to use the route attributes on the controller level as well as on the individual action methods level.

Let us understand this with an example. First, modify the Home Controller class as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"Home"**)]**

**[**Route**(**"Home/Index"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**[**Route**(**"Home/Details/{id?}"**)]**

**public** string Details**(int** id**)**

**{**

**return** "Details() Action Method of HomeController, ID Value = " + id;

**}**

**}**

**}**

With the above code in place, we can access the Index() action method using the following 3 URLs.  
**/**  
**/Home**  
**/Home/Index**  
Along the same line, we can also access the Details(int? id) action method using the following 2 URLs.   
**/Home/Details**  
**/Home/Details/2**

If you notice, we have repeated the word Home multiple times (four times in our example). In order to make these routes less repetitive, we need to apply the Route() attribute with the word Home at the HomeController class level as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**[**Route**(**"Home"**)]**

**public** **class** HomeController : Controller

**{**

**[**Route**(**""**)]**

**[**Route**(**"Index"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**[**Route**(**"Details/{id?}"**)]**

**public** string Details**(int** id**)**

**{**

**return** "Details() Action Method of HomeController, ID Value = " + id;

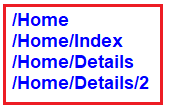
**}**

**}**

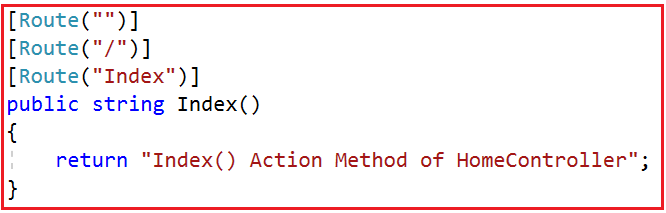
**}**

**Note:** The Route template applied on the controller level is prepended to the route template applied to the action method level.

Now when you navigate to the following four URLs you will get the output as expected.



However, when you navigate to the root URL (**http://localhost:52190**) of the application, you will get a 404 error. In order to solve this, you need to include the route template that begins with **“/”** on the Index() action method as shown below.

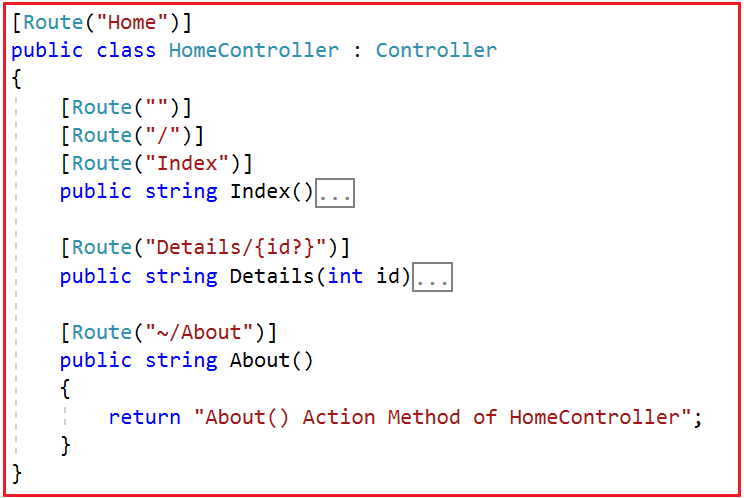


With the above changes in place, now run the application and navigate to the root URL and you will see the output as expected.

**How to ignore the Route Template placed at the Controller Level?**

In order to ignore the Route Template placed at the Controller level, you need to use / or ~/ at the action method level. If the action method route template starts with **/ or ~/**, then the controller route template is not going to be combined with the action method route template.

To understand this let us modify the Home Controller class as shown below. In the following code, the About action method starts with ~/, so this action method is not going to be combined with the controller route template.



Now run the application and navigate to /About URL and you will see the output as expected.

**ASP.NET Core Attribute Routing using Tokens**

**ASP.NET Core Attribute Routing using Tokens**

In this article, I am going to discuss the **ASP.NET Core Attribute Routing using Tokens** with examples. Please read our previous article before proceeding to this article as we are going to work with the same example that we worked in our previous article. In our previous article, we discussed [**Attribute Routing in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/attribute-routing-in-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers.

1. **Understanding Tokens in Attribute Routing.**
2. **Token Example in Attribute Routing.**
3. **Advantages of using Tokens in Attribute Routing.**
4. **Do we need to write the action token on each action method?**
5. **Attribute Routing vs Conventional Routing in ASP.NET Core.**

**Tokens in Attribute Routing:**

In ASP.NET Core, the Route Attribute support token replacement. It means we can enclose the token (i.e. controller and action) within a pair of square-braces (**[ ]**). The tokens (i.e. **[controller]** and **[action]**) are then replaced with the values of controller and action method name where the route is defined.

**Token Example in Attribute Routing:**

Let us understand this with an example. Please modify the Home Controller class as shown below. Here we are applying the token **[controller]** on the Home Controller and at the same time, we are applying the token **[action]** on all the action methods.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**[**Route**(**"[controller]"**)]**

**public** **class** HomeController : Controller

**{**

**[**Route**(**"[action]"**)]**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**[**Route**(**"[action]"**)]**

**public** string Details**()**

**{**

**return** "Details() Action Method of HomeController";

**}**

**}**

**}**

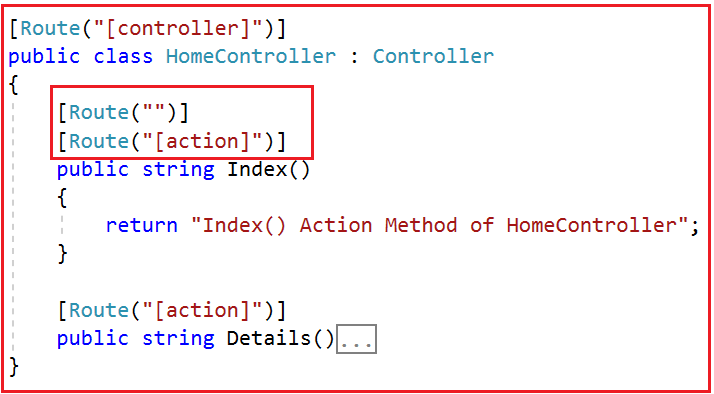
With the above controller and action tokens in place, now you can access the Index action method of Home Controller with the URL **/Home/Index**. Similarly, you can access the Details action method using the URL **/Home/Details**. Now run the application and see everything is working as expected.

**Advantages of Tokens in Attribute Routing:**

The main advantage is that in the future if you rename the controller name or the action method name then you do not have to change the route templates. The application is going to works with the new controller and action method names.

**How to make the Index action method as the default action?**

With the controller and action tokens in place, if you want to make the Index action method of Home Controller as the default action, then you need to include the **Route(“”)** attribute with an empty string on the Index action method as shown below.



**Do we need to write the action token on each action method?**

Not Really. If you want all your action methods to apply action token, then instead of including the [action] token on each and every action method, you can apply it only once on the controller as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**[**Route**(**"[controller]/[action]"**)]**

**public** **class** HomeController : Controller

**{**

**public** string Index**()**

**{**

**return** "Index() Action Method of HomeController";

**}**

**public** string Details**()**

**{**

**return** "Details() Action Method of HomeController";

**}**

**}**

**}**

**Attribute Routing vs Conventional Routing in ASP.NET Core:**

In Attribute Routing, we need to define the routes using the Route attribute within the controller and action methods. The Attribute routing offers a bit more flexibility than conventional based routing. However, in general, the conventional based routings are useful for controllers that serve HTML pages. On the other hand, the attribute routings are useful for controllers that serve RESTful APIs.

However, there is nothing stopping you from mixing conventional based routing with attribute routing in a single application.

# Layout View in ASP.NET Core MVC

## ****Layout View in ASP.NET Core MVC****

In this article, I am going to discuss the **Layout View in ASP.NET Core MVC** Application. Please read our previous article where we discussed the [**Attribute Routing in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/attribute-routing-in-asp-net-core-mvc/) Application with examples. As part of this article, we are going to discuss the following pointers.

1. **Why do we need Layout View in ASP.NET Core MVC?**
2. **What is a Layout?**
3. **How to Create a Layout View in ASP.NET Core MVC Application?**
4. **How to use a Layout View in ASP.NET Core MVC Application?**

##### ****What is Layout?****

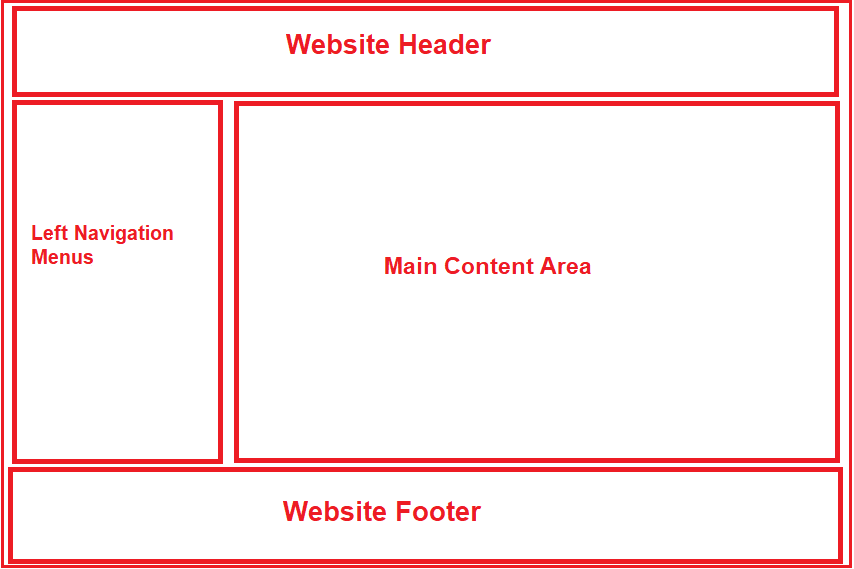
The layouts are like the master pages in Webforms applications.  The common UI code, which can be used in many views can go into a common view called layout.

##### ****Why do we need Layout View in ASP.NET Core MVC?****

Nowadays, almost all web applications have the following sections.

1. **Website Header**
2. **Website Footer**
3. **Navigation Menus**
4. **Main Content Area**

Please have a look at the following image which shows the above mentioned four areas on a website.



If you don’t have a layout view for your website, then you need to repeat the required HTML for the above-mentioned sections in each and every view of your application. This is violating the DRY (Don’t Repeat Yourself) principle as we are repeating the same code in multiple views. As a result, it is very difficult to maintain the application. For example, if you have to remove or add a menu item from the list of navigation menus or even if you want to change the header or footer of your website then you need to do this in each and every view which is tedious, time-consuming as well as error-prone.

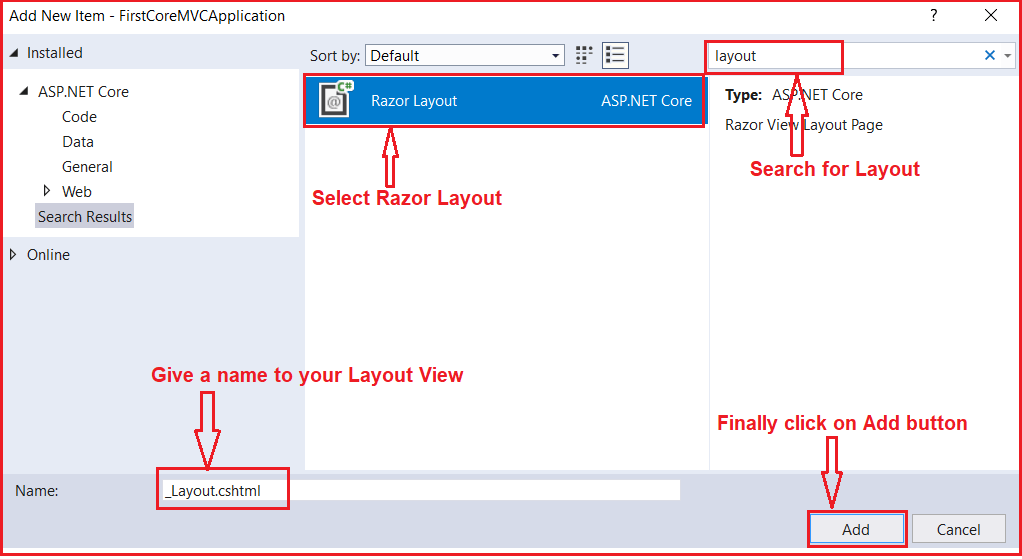
Instead of putting all the sections (i.e. the HTML) in each and every view pages, it is always better and advisable to put them in a layout view and then inherit that layout view in each and every view where you want that look and feel. With the help of layout views, now it is easier to maintain the consistent look and feel of your application. This is because if you at all need to do any changes then you need to do it only at one place i.e. in the layout view and the changes will be reflected immediately across all the views which are inherited from the layout view.

##### ****Layout View in ASP.NET Core MVC Application:****

1. Like the regular view in ASP.NET Core MVC, the layout view is also a file with a .cshtml extension
2. If you are coming from ASP.NET Web Forms background, you can think the layout view as the master page in asp.net web forms application.
3. As the layout views are not specific to any controller, so, we usually place the layout views in a subfolder called “**Shared**” within the “**Views**” folder.
4. By default, in ASP.NET Core MVC Application, the layout view file is named **\_Layout.cshtml**.
5. The leading underscore in the file name indicates that these files are not intended to be served directly by the browser.
6. In ASP.NET Core MVC, it is also possible to create multiple layout files for a single application. For example, you may have one layout file for the admin users and another layout file for non-admin users of your application.

**How to Create a Layout View in ASP.NET Core MVC Application?**In order to create a layout view in ASP.NET Core MVC, you need to follow the below steps.

1. Right-click on the “**Views**” folder and then add a new folder with the name “**Shared**“.
2. Next, Right-click on the “**Shared**” folder and then select the “**Add**” – “**New Item**” option from the context menu which will open the Add New Item window.
3. From the “**Add New Item**” window search for **Layout** and then select “**Razor Layou**t”, give a meaning full name (\_Layout.cshtml) to your layout view and finally click on the “**Add**” button as shown below which should add **\_Layout.cshtml** file within the Shared folder.



Note: In this article, I am going to show you how to create and use a layout file and in our upcoming articles, I will show you how to use website header, footer, and navigation menus.

##### ****Understanding the \_Layout.cshtml file:****

Let us have a look at the auto-generated HTML code in the **\_Layout.cshtml** file. The following HTML is auto-generated in the **\_Layout.cshtml** file.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<div>**

@RenderBody()

**</div>**

**</body>**

**</html>**

As you can see in the above layout file, it contains the standard Html, head, title and body elements. As the above elements are present in the layout file, so you don’t have to repeat all the above elements in each and every view.

The View or Page-specific title is retrieved by using the **@ViewBag.Title** expression. For example, when “**index.cshtml**” view is rendered using this layout view, then the **index.cshtml** view will set the Title property on the ViewBag. This is then retrieved by the Layout view using the expression **@ViewBag.Title** and set as the value for the <title> tag.

The **@RenderBody()** specifies the location where the view or page-specific content is injected. For example, if “**index.cshtml**” view is rendered using this layout view, then **index.cshtml** view content is injected at the location.

Let us modify the **\_Layout.cshtml** page as shown below to include the header, footer, left navigation menus and main content area section.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<table** border="1" style="width:800px; font-family:Arial"**>**

**<tr>**

**<td** colspan="2" style="text-align:center"**>**

**<h3>**Website Header**</h3>**

**</td>**

**</tr>**

**<tr>**

**<td** style="width:200px"**>**

**<h3>**Left Navigation Menus**</h3>**

**</td>**

**<td** style="width:600px"**>**

@RenderBody()

**</td>**

**</tr>**

**<tr>**

**<td** colspan="2" style="text-align:center; font-size:x-small"**>**

**<h3>**Website Footer**</h3>**

**</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

##### ****Modifying the Startup class:****

Please modify the Startup class as shown below where we configure the required services for MVC.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *FirstCoreMVCApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

app.UseMvcWithDefaultRoute**()**;

**}**

**}**

**}**

##### ****Modifying the Home Controller:****

Please modify the Home Controller as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**()**;

**}**

**public** ViewResult Details**()**

**{**

**return** View**()**;

**}**

**}**

**}**

As you can see here we have created two action methods i.e. Index and View.

##### ****Using Layout view in ASP.NET Core MVC Application:****

Now we are going to create the Index and Details views using the Layout view. In order to render a view using the layout view (\_Layout.cshtml), you need to set the Layout property.

###### **Index.cshtml:**

Please modify the Index view as shown below to use the layout view.

@{

ViewBag.Title = "Home Page";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<h1>**Home Page**</h1>**

###### **Details.cshtml:**

Please modify the Details view as shown below to use the layout view.

@{

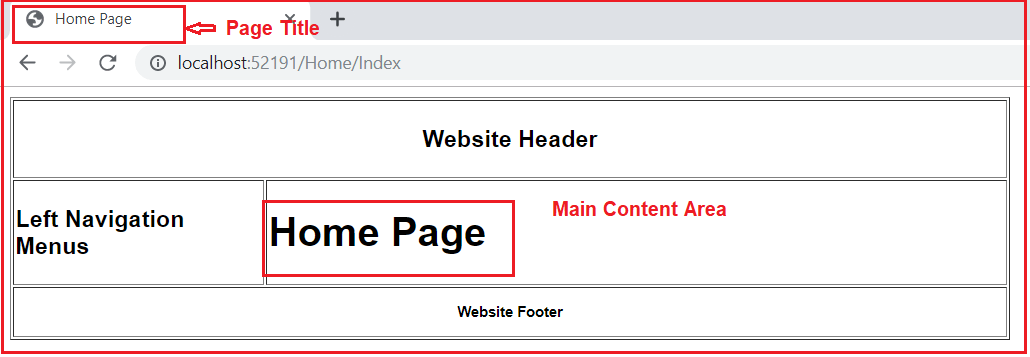
ViewBag.Title = "Details Page";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<h1>**Details Page**</h1>**

Now run the application and navigate to the Home/Index URL which should display the page as shown below.



**Sections in Layout Page in ASP.NET Core MVC**

**Sections in Layout Page in ASP.NET Core MVC**

In this article, I am going to discuss the **Sections in Layout Page in ASP.NET Core MVC** Application. Please read our previous article where we discussed the [**Layout View in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/layout-view-in-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers.

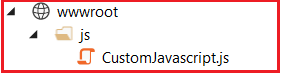
1. **What are Sections?**
2. **What is the need for Sections in Layout View in ASP.NET Core MVC Application?**
3. **Understanding the RenderSection Method.**
4. **How to use the RenderSection Method in ASP.NET Core MVC?**
5. **How to Provide section Content in a View?**
6. **Understanding How to make the layout section optional in ASP.NET Core MVC?**

**Sections in Layout View in ASP.NET Core MVC Application:**

In ASP.NET Core MVC, the layout view contains one or more sections in it. The Sections in a layout view are used to organize where certain page elements should be placed. The sections in a layout view can be optional or mandatory.

**Understanding the need for Sections with an Example:**

In order to understand this, let us first create a custom javascript file. First, create a folder at the root level of the application with the name “**wwwroot”**. In generally all the static files we need to be placed within this folder. Once you created the “**wwwroot**” folder create a subfolder within this with the name **“js”** and then add a javascript file with the name **“CustomJavascript.js”** within the js folder as shown in the below image.



**Situation1:**

If we have a custom javascript file (i.e. CustomJavascript.js) and that file is being required by all the views of our application, then we need to place it in the Layout View of our application as shown below.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<table** border="1" style="width:800px; font-family:Arial"**>**

**<tr>**

**<td** colspan="2" style="text-align:center"**>**

**<h3>**Website Header**</h3>**

**</td>**

**</tr>**

**<tr>**

**<td** style="width:200px"**>**

**<h3>**Left Navigation Menus**</h3>**

**</td>**

**<td** style="width:600px"**>**

@RenderBody()

**</td>**

**</tr>**

**<tr>**

**<td** colspan="2" style="text-align:center; font-size:x-small"**>**

**<h3>**Website Footer**</h3>**

**</td>**

**</tr>**

**</table>**

**<script** src="~/js/CustomJavascript.js"**></script>**

**</body>**

**</html>**

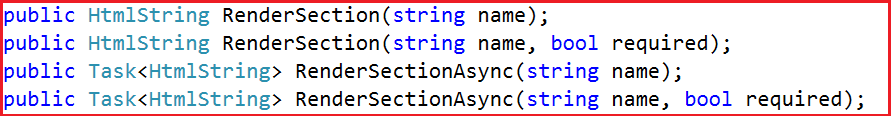
**Note:** It is always a good programming practice to put all the script files before the closing body tag.

**Situation2:**

If you have a custom javascript file (i.e. CustomJavascript.js) and you want that file in some specific views. Let assume you want that file in the Index view but not in the Details view of Home Controller. In such scenarios, you can make use of the section.

**Understanding the RenderSection Method:**

Let us have a look at the signature of the RenderSection() method which is shown below.



As you can see there are two overloaded versions of the **RenderSection** Method. The same is the case for the **RenderSectionAsync** method. The first version of the Render section method takes a single parameter (i.e. name) which specifies the name of the section. The second overloaded version takes two parameters. The first parameter (name) specifies the name of the section while the second parameter (required) specifies whether the section is required or optional.

**How to use the RenderSection Method in ASP.NET Core MVC?**

In your layout view, you need to call the **RenderSection()** method at the location where you want the section content to be rendered. In our example, we want the script file to be included just before the closing **</body>** tag. So, we are calling the **@RenderSection()** method just before the closing **</body>** tag as shown below.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<table** border="1" style="width:800px; font-family:Arial"**>**

**<tr>**

**<td** colspan="2" style="text-align:center"**>**

**<h3>**Website Header**</h3>**

**</td>**

**</tr>**

**<tr>**

**<td** style="width:200px"**>**

**<h3>**Left Navigation Menus**</h3>**

**</td>**

**<td** style="width:600px"**>**

@RenderBody()

**</td>**

**</tr>**

**<tr>**

**<td** colspan="2" style="text-align:center; font-size:x-small"**>**

**<h3>**Website Footer**</h3>**

**</td>**

**</tr>**

**</table>**

@RenderSection("Scripts")

**</body>**

**</html>**

In the above code, we are using the first overloaded version of the RenderSection method which takes only the name parameter.

**How to Provide section Content in a View?**

In our layout view, we have created a section. Now let us understand how to provide section content in a view. Each and every view which wants to provide section content must include a section within the same. Here, you need to use the @section directive to include the section and provide the content.

In our example, we want to provide the section content from the Index view. So, modify the index view as shown below.

@{

ViewBag.Title = "Home Page";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<h1>**Home Page**</h1>**

@section Scripts {

**<script** src="~/js/CustomJavascript.js"**></script>**

}

**Modify the Startup class as shown below.**

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *FirstCoreMVCApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

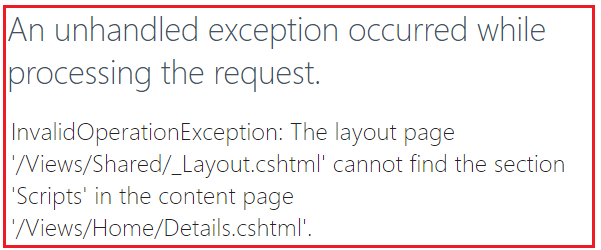
app.UseMvcWithDefaultRoute**()**;

**}**

**}**

**}**

Now run the application and navigate to Home/Index URL and you will see the out as expected. But, when you navigate to Home/Details URL, you will get the following error page.



The reason for getting the above exception is the section is mandatory and we have not specified the section content in the Details view.

**How to make the layout section optional in ASP.NET Core MVC?**

We can make a layout section optional in ASP.NET Core MVC in two ways. They are as follows:

**Way1:**Use the RenderSection method which takes two parameters. Set the second parameter (i.e. the required) to false.

**@RenderSection(“Scripts”, required: false)**

**Way2:**Using the IsSectionDefined() method. This method returns a value that indicates whether the specified section is defined in the content page.

**@if (IsSectionDefined(“Scripts”))**  
**{**  
**@RenderSection(“Scripts”, required: false)**  
**}**

So, modify the **\_Layout.cshtml** file as shown below to make the section as optional.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<table** border="1" style="width:800px; font-family:Arial"**>**

**<tr>**

**<td** colspan="2" style="text-align:center"**>**

**<h3>**Website Header**</h3>**

**</td>**

**</tr>**

**<tr>**

**<td** style="width:200px"**>**

**<h3>**Left Navigation Menus**</h3>**

**</td>**

**<td** style="width:600px"**>**

@RenderBody()

**</td>**

**</tr>**

**<tr>**

**<td** colspan="2" style="text-align:center; font-size:x-small"**>**

**<h3>**Website Footer**</h3>**

**</td>**

**</tr>**

**</table>**

@RenderSection("Scripts", false)

**</body>**

**</html>**

With the above changes in place, run the application and navigate to both the URL and you should get the output as expected.

**ViewStart in ASP.NET Core MVC**

**ViewStart in ASP.NET Core MVC Application**

In this article, I am going to discuss the **ViewStart in ASP.NET Core MVC** Application. Please read our previous article before proceeding to this article as it is a continuation part of our previous article. In our previous article, we discussed the [**Sections in the Layout Page in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/sections-in-layout-view-in-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers.

1. **Why do we need \_ViewStart.cshtml file in ASP.NET Core MVC Application?**
2. **What is \_ViewStart.cshtml file in ASP.NET Core MVC Application?**
3. **How to create \_ViewStart.cshtml file in ASP.NET Core MVC Application?**
4. **How to set the Layout Property in ViewStart.cshtml file?**
5. **Understanding the hierarchical of \_ViewStart.cshtml file.**
6. **How to select a layout conditionally in ViewStart file?**

**Why do we need \_ViewStart.cshtml file in ASP.NET Core MVC Application?**

As of now, we have used the Layout Property to associate a view with a layout view as shown below.

**Layout = “~/Views/Shared/\_Layout.cshtml”;**

Suppose, we have 100 views in our application and all the 100 views want to use the same layout file. Then we need to set the Layout Property as shown in the above image in all the 100 views. This violates the **DRY (Don’t Repeat Yourself)** Principle and has the following disadvantages.

1. **Redundant Code**
2. **Maintenance Overhead**

Suppose tomorrow you want to use a different Layout, then you need to update the Layout Property in each and every individual view. This process is tedious, time-consuming as well as error-prone because you may miss updating the Property in some of the views. To solve the above problems we need to use the \_ViewStart.cshtml file.

**What is \_ViewStart.cshtml file in ASP.NET Core MVC Application?**

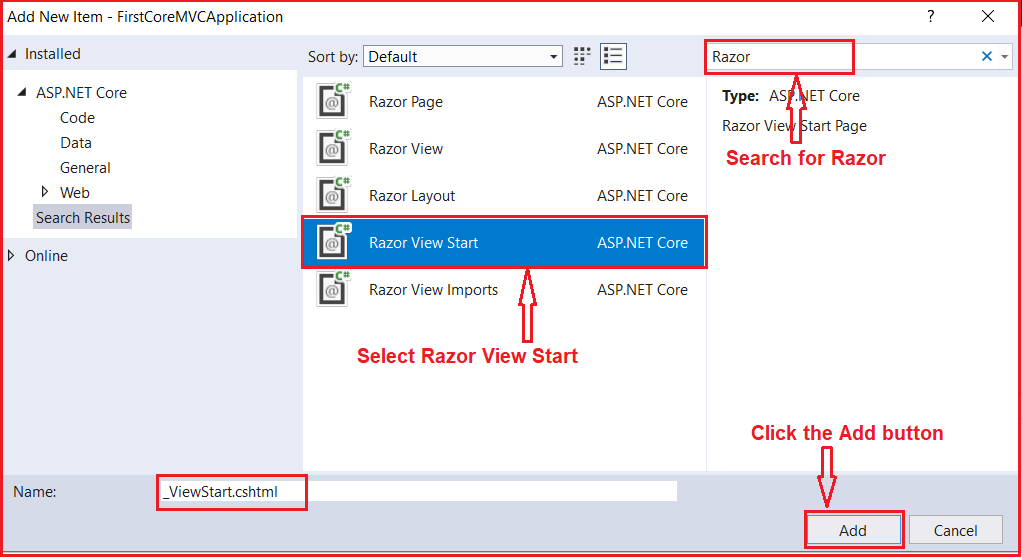
In ASP.NET Core MVC Application, the **\_ViewStart.cshtml** file is a special file and the code present in this file is going to be executed before the code in an individual view is executed. So, you can set the Layout Property in this file as shown in the below image instead of setting the Layout Property in each individual view which is going to be executed before the actual view is executed.



Once you set the **Layout** Property in **\_ViewStart.cshtml** file as shown in the above image, then maintaining the application becomes much easier. So, in the future, if you want to change the layout file, then you just need to change the code at one place only i.e. in the \_ViewStart.cshtml file.

**How to create \_ViewStart.cshtml file in ASP.NET Core MVC Application?**

In general, the \_**ViewStart.cshtml**files are created within the Views or within the subfolder of the Views folder. To create **“\_ViewStart.cshtml”** file right click on the Views folder and then select **“Add – New Item”** option from the context menu, this will opens the **“New Item”** window. From the **“New Item”** window search for **“Razor”** and then select the **“Razor View Start”** and click on the **“Add”** button as shown in the below image which should create the **“\_ViewStart.cshtml”** within the **“Views”** folder.



**How to set the Layout Property in ViewStart.cshtml file?**

Once the **ViewStart.cshtml** file is created then modify the file as shown below to set the **Layout** property.

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

Then we need to remove the Layout property on individual views. So, modify the Index view as shown below.

@{

ViewBag.Title = "Home Page";

}

**<h1>**Home Page**</h1>**

@section Scripts {

**<script** src="~/js/CustomJavascript.js"**></script>**

}

With the above changes in place, now run the application and it should display the output as expected.

**Understanding the hierarchical of \_ViewStart.cshtml file:**

As we already discussed we can place the ViewStart file Views folder and its subfolder. So, we need to understand the hierarchical order of the ViewStart file. Let us understand this with an example.

Let us first create another layout file with the name **\_MyLayout.cshtml** within the shared folder. Once you create the **\_MyLayout.cshtml** file, then copy and paste the following code.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**</head>**

**<body>**

**<div>**

@RenderBody()

**</div>**

@if (IsSectionDefined("Scripts"))

{

@RenderSection("Scripts", false)

}

**</body>**

**</html>**

With this Layout, now we have two layouts (**\_Layout.cshtml and \_MyLayout.cshtml**) for our application.

**Creating ViewStart File within the Home Folder:**

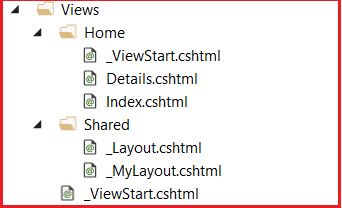
Let add another **ViewStart** file within the Home Folder which is present within the Views folder. Once you create the **ViewStart** file then modify the file as shown below. Here, we are setting the newly created **\_MyLayout.cshtml** in the Layout property.

@{

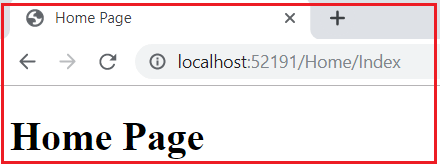
Layout = "~/Views/Shared/\_MyLayout.cshtml";

}

With the above changes, the Views folder of your application should looks as shown below.



As you can see in the above image, we have placed one **ViewStart**file in the **Views**folder and another **ViewStart**file in the **Home**sub-folder. Now run the application and see the output.



The above Index view uses MyLayout.cshtml view which we specified within the ViewStart File which is present inside the Home Folder. So, here the layout page which is specified in the ViewStart file in the Home sub-folder overwrites the layout page specified in the ViewStart file in the Views folder.

This means, all the views which are present within the **Views**folder will use the layout page which is specified in the ViewStart file in the Views folder, but the views which are present in the Home folder will use the layout page which is specified in the ViewStart file in the Home folder.

**Note:**If you don’t want to use the layout file which is specified in the ViewStart file, rather you want to use a different layout file then you need to use the Layout property in individual view to set the layout. If you don’t want to use any layout or if you want to render a view without layout then need to set the Layout property to null.

**How to select a layout conditionally in ViewStart file?**

In an ASP.NET Core MVC application, you may have multiple layout views. Let’s say you have two Layouts such as \_NonAdminLayout and \_AdminLayout. If you want to select the Layout based on the user role i.e. if the user role is Admin then use \_AdminLayout else use the \_NonAdminLayout. Then you need to write the following logic within the \_ViewStart.cshtml file which will select the layout based on the role of the logged-in user.

@{

if (User.IsInRole("Admin"))

{

Layout = "\_AdminLayout";

}

else

{

Layout = "\_NonAdminLayout";

}

}

**ViewImports in ASP.NET Core MVC**

**ViewImports in ASP.NET Core MVC**

In this article, I am going to discuss the **ViewImports in ASP.NET Core MVC** Application. Please read our previous article before proceeding to this article as it is a continuation part of our previous article where we discussed the [**ViewStart in Layout Page in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/viewstart-in-asp-net-core-mvc/) Application. The ASP.NET Core MVC and Razor comes with a lot of new advanced features for working with the Razor views. ViewImports is one of the new features. As part of this article, we are going to discuss the following pointers.

1. **What is \_ViewImports.cshtml?**
2. **Understanding ViewImports with an example.**
3. **Creating ViewImports.cshtml file in ASP.NET Core MVC Application.**
4. **Understanding the hierarchical Order of ViewImports file in ASP.NET Core MVC.**

**What is \_ViewImports.cshtml in ASP.NET Core MVC Application?**

In ASP.NET Core MVC Application, the **\_ViewImports.cshtml** file provides a mechanism to include the directives globally for Razor Pages so that we don’t have to add them individually in each and every page. As of this article, the \_ViewImports.cshtml file supports the following directives:

1. **@addTagHelper**
2. **@tagHelperPrefix**
3. **@removeTagHelper**
4. **@namespace**
5. **@inject**
6. **@model**
7. **@using**

The **@addTagHelper**, **@tagHelperPrefix,**and **@removeTagHelper** directives are basically used to manage of Tag Helpers. The **@namespace** directive is basically used to specify the namespace that the **ViewImports** belongs to. With the help of **@inject** directive, it supports Dependency injection. We already use the **@model** directive in our previous applications when we are working with models. The **@model** directive is basically used to specify the Model for your view. The **@using** directive basically used to include the common namespaces globally so that you don’t have to include the namespaces in each and every view page.

**Note:** In this article, I am going to show you the use of the @using directive in ViewImports.cshtml file. Rest directives are going to be discussing in our upcoming articles.

**Let us understand ViewImports with an example:**

Create a model called Students within the Models folder of your application. Once you create the **Students.cs** class file then copy and paste the following code in it.

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

As you can see, here we created the student model with five properties.

**Modifying the Home Controller:**

Modify the Home Controller as shown below.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 102, Name = "Smith", Branch = "ETC", Section = "B", Gender = "Male" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 104, Name = "Sara", Branch = "CSE", Section = "A", Gender = "Female" **}**,

new Student**()** **{** StudentId = 105, Name = "Pam", Branch = "ETC", Section = "B", Gender = "Female" **}**

**}**;

**return** View**(**listStudents**)**;

**}**

**public** ViewResult Details**(int** Id**)**

**{**

var studentDetails = new Student**()** **{** StudentId = Id, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**;

**return** View**(**studentDetails**)**;

**}**

**}**

**}**

As you can see, here we have two action methods. One action method is used to display all the student data while the other action method takes the student id as a parameter and return that student information.

**Modifying the Index and Details view:**

**Index.cshtml:**

@model List**<FirstCoreMVCApplication.Models.Student>**

@{

Layout = null;

}

**<html>**

**<head>**

**<title>**Index**</title>**

**</head>**

**<body>**

**<table>**

**<thead>**

**<tr>**

**<th>**ID**</th>**

**<th>**Name**</th>**

**<th>**Branch**</th>**

**<th>**Section**</th>**

**<th>**Gender**</th>**

**</tr>**

**</thead>**

**<tbody>**

@foreach (var student in Model)

{

**<tr>**

**<td>**

@student.StudentId

**</td>**

**<td>**

@student.Name

**</td>**

**<td>**

@student.Branch

**</td>**

**<td>**

@student.Section

**</td>**

**<td>**

@student.Gender

**</td>**

**</tr>**

}

**</tbody>**

**</table>**

**</body>**

**</html>**

**Details.cshtml:**

@model FirstCoreMVCApplication.Models.Student

@{

Layout = null;

}

**<html>**

**<head>**

**<title>**Student Detaills**</title>**

**</head>**

**<body>**

**<div>**

StudentId : @Model.StudentId

**</div>**

**<div>**

Name : @Model.Name

**</div>**

**<div>**

Branch : @Model.Branch

**</div>**

**<div>**

Section : @Model.Section

**</div>**

**<div>**

Gender : @Model.Gender

**</div>**

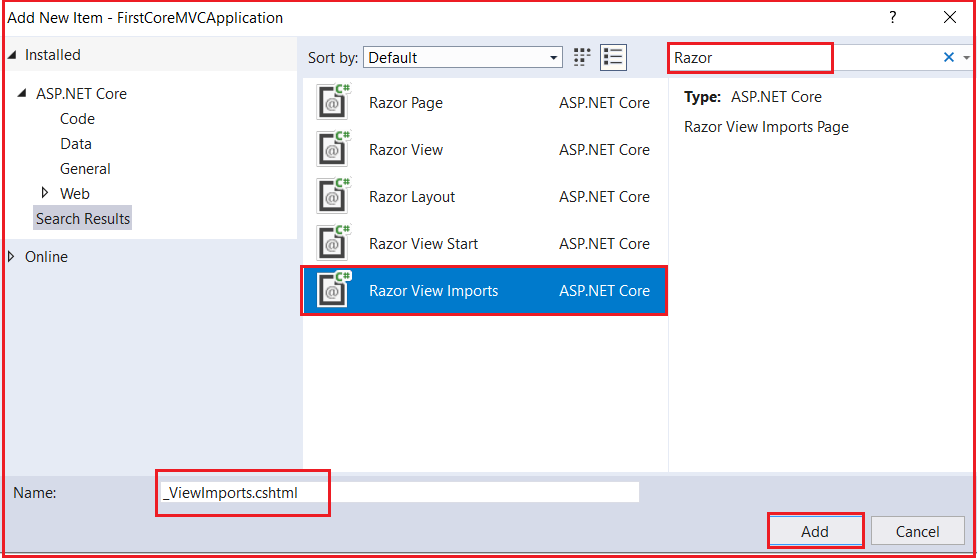
**</body>**

**</html>**

In the above Index and Details view, we are using the **@model** directive to specify the model for the view. If you notice, then you can see in both the views we have specified the fully qualified name for the model such as “**FirstCoreMVCApplication.Models.Student**”. Now let us see how to move the namespace to the **ViewImports** file so that we can only specify the model name.

**Creating ViewImports.cshtml file in ASP.NET Core MVC Application:**

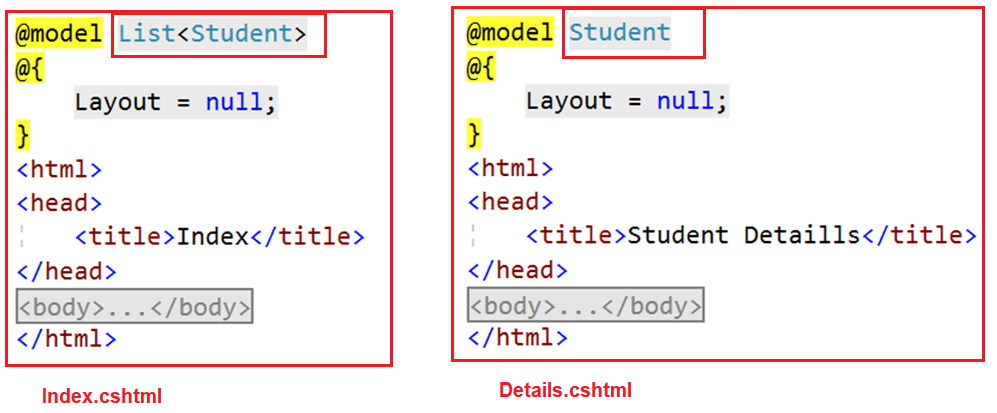
In general, \_**ViewImports.cshtml**files are created within the Views or within the subfolder of the Views folder. To create the **“\_ViewImports.cshtml”** file right-click on the Views folder and then select the **“Add – New Item”** option from the context menu, which will open the **“New Item”** window. From the **“New Item”** window search for **“Razor”** and then select the **“Razor View Import”** and click on the **“Add”** button as shown in the below image which should create the **“\_ViewImport.cshtml”** within the **“Views”** folder.



Once the \_ViewImports.cshtml file is created, then copy and paste the following code in it.

**@using FirstCoreMVCApplication.Models;**

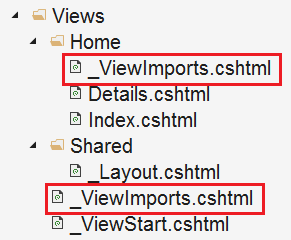
As we placed the above namespace in the **ViewImports** file, now all the types that are present in the above namespace are available to each and every view in the **“Home”** folder. So now we don’t need to type the fully qualified name of the Type. So, modify the Index and Details view as shown below.



As you can see in the above image, we are removing the namespace and only specified the model name. Run the application and it should work as expected.

**\_ViewImports file is hierarchical Order in ASP.NET Core MVC:**

Just like the **\_ViewStart** file, the **\_ViewImports** file is also hierarchical. It is also possible to pace the \_ViewImports in the subfolder of the Views folder as shown in the below image. Here we have one **\_ViewImports** file in the Views folder and another **\_ViewImports** file within the Home folder.



The settings that are specified in the \_ViewImports file present in the **Home**subfolder will overwrite the settings specified in the **\_ViewImports** file in the Views folder.

**Note:**If you specified a setting in the view itself, then that setting overrides the matching settings specified in the parent \_ViewImports files in the folder hierarchy.

**How to Install Bootstrap in ASP.NET Core MVC**

**How to Install Bootstrap in ASP.NET Core MVC Application**

In this article, I am going to discuss **How to Install Bootstrap in ASP.NET Core MVC** Application. Please read our previous article, where we discussed the [**ViewImports in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/viewimports-in-asp-net-core-mvc/) Application. As part of this article, I am going to discuss the following pointers.

1. **Different Tools to Install Client-Side Packages in ASP.NET Core.**
2. **What is Library Manager or Libman in ASP.NET Core?**
3. **How to Check and Upgrade the Version in Visual Studio?**
4. **How to Install Bootstrap in ASP.NET Core Using Library Manager?**
5. **What is libman.json file in ASP.NET Core?**
6. **How to Clean and Restore Client-Side Libraries using Libman in ASP.NET Core?**
7. **How to uninstall or update a Client-Side Library using libman.json file?**

**Different Tools to Install Client-Side Packages in ASP.NET Core:**

There are many tools available that you can use to install client-side packages such as JQuery and Bootstrap using Visual Studio. Some of the popular tools are as follows/;

1. **Bower**
2. **NPM**
3. **WebPack, etc.**

But here in this article, I am not going to use any of the above tools instead we are going to use Library Manager which is known as Libman to install the client-side packages. In our upcoming article, I will show you how to use Bower, NPM, and WrebPack to install the client-side packages.

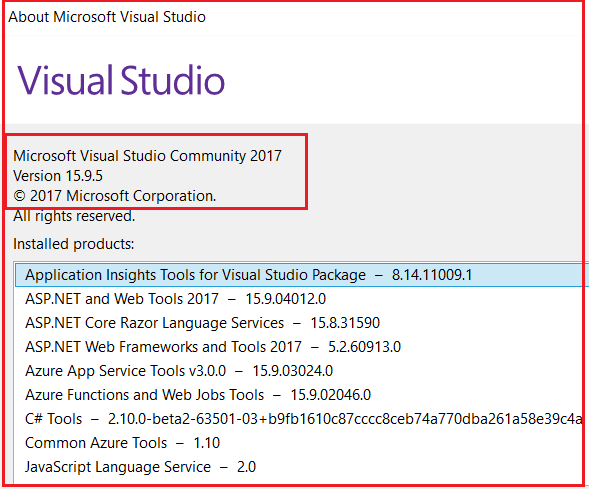
**What is Library Manager or Libman in ASP.NET Core?**

The Library Manager or LinMan is one of the most popular light-weight, client-side library acquisition tool. This tool is basically used to download the client-side libraries and frameworks such as Bootstrap and JQuery from a file system or from a CDN (Content Delivery Network). In order to use Library Manager, you should have Visual Studio 2017 version 15.8 or later.

**How to Check and Upgrade the Version in Visual Studio?**

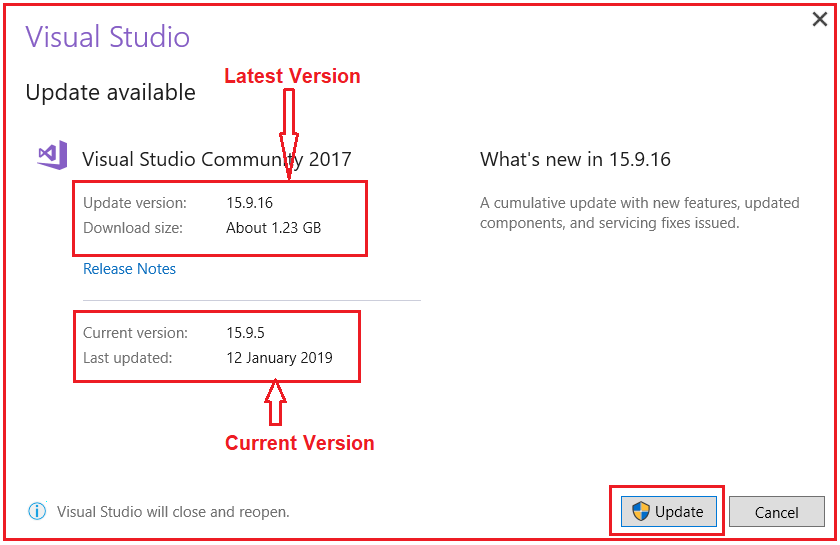
In order to check the Visual Studio Version, you need to follow the below steps.

Click on the “**Help**” menu and then select “**About Microsoft Visual Studio**” option from the context menu. This will open the “**About Microsoft Visual Studio**” window which shows the version number of Visual Studio as shown in the below image. On my machine, I have installed 15.9.5.



If you have installed an older version of Visual Studio 2017, then you can easily update it. In order to update the version of visual studio, you need to follow the below steps.

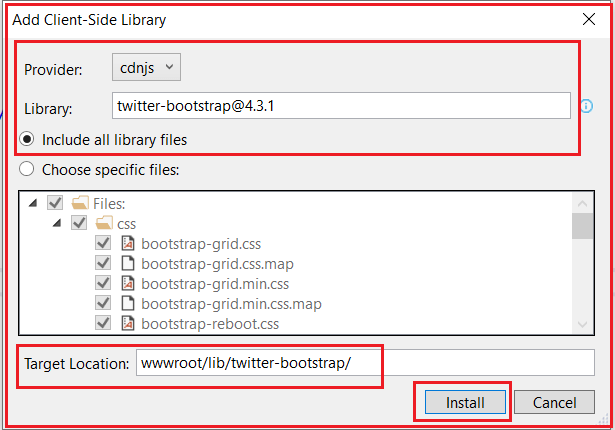
Click on the “**Help**” menu and then select the “**Check for Updates**” option from the context menu. The window appears will display the current version as well as the latest version available of visual studio. Then you need to click on the “**Update**” button as shown in the below image which will update your visual studio 2017 to its latest version.



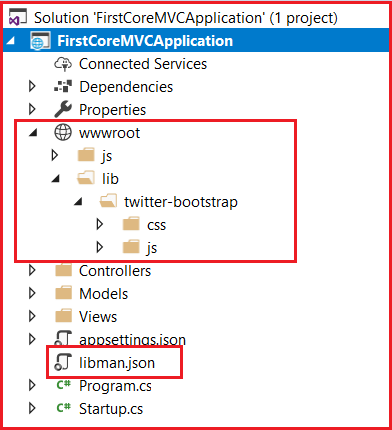
**How to Install Bootstrap in ASP.NET Core Using Library Manager?**

You need to follow the below steps to install Bootstrap in ASP.NET Core MVC Application using the Library Manager (Libman).

1. Right-click on the “**Project Name**” in the Solution Explorer and then select “**Add > Client-Side Library”** which will open “**Add Client-Side Library**” window.
2. Leave the default provider as it is which “**cdnjs**” is in this case. The other providers are **filesystem** and **unpkg**.
3. In the “**Library**” text box, just type “**twitter-bootstrap**“. You can also get intelligence support once you start typing. Once you select the matching entry, then it tries to install the latest version of bootstrap. However, if you want then you can also type the version manually you want. Here, we are installing the latest version of Bootstrap i.e. (“**twitter-bootstrap@4.3.1**”).
4. There are two radio buttons to select whether you to include “**All library files**” or “**Choose Specific files**“. If you select “**All library files**” radio button then all the files are going to be downloaded. On the other hand, if you select “**Choose Specific files**” radio button then you need to check the selected checkboxes as per your requirement. Here I am selecting the “**All library files**” radio button.
5. In the “**Target Location**” text box specify the folder location where you want the library files to be installed. By default, the static files are only served from the **wwwroot** folder. Here, I am going with the default location i.e. “**wwwroot/lib/twitter-bootstrap/”**.
6. Finally, click on the “**Install**” button as shown in the image below.



Once it successfully installed, then you will find two things. One is libman.json file and the second one the required bootstrap files. Please have a look at the following image.



**What is libman.json file in ASP.NET Core?**

The **libman.json** file is the Library Manager manifest file. You will find the following code in the libman.json file.



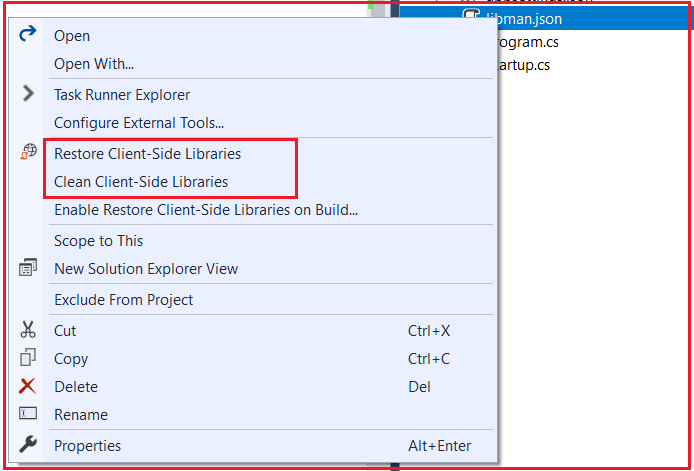
As you can see in the above image, we have an entry for the Bootstrap library that we just installed using the Libman. It is also possible to install the client-side libraries like bootstrap and JQuery by editing the above manifest file.

**How to clean Client-Side Libraries using Libman in ASP.NET Core?**

If you want to clean the library files which are created by using the Library Manager, then you just need to right-click on the **libman.json** file and then select the “**Clean Client-Side Libraries**” option from the context menu. Once you click on the “**Clean Client-Side Libraries**” option then it will delete all the library files from the respective destination folder. The point that you need to keep in mind is, it will only delete the files from the folder but not in the libman.json file.

**How to Restore Client-Side Libraries using Libman in ASP.NET Core?**

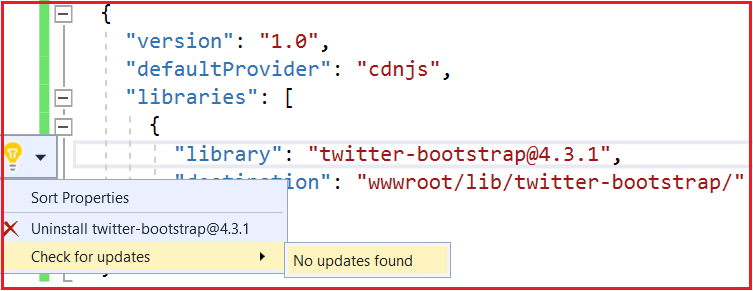
If you want to restore the deleted files, then you just need to right-click on libman.json file and then select the **“Restore Client-Side Libraries”** option from the context menu. This will again download and installed the required library files into the specified destination folder.



**How to uninstall or update a Client-Side Library using libman.json file?**

If you want to uninstall or update a client-side library using libman.json file, then you need to follow the below steps.

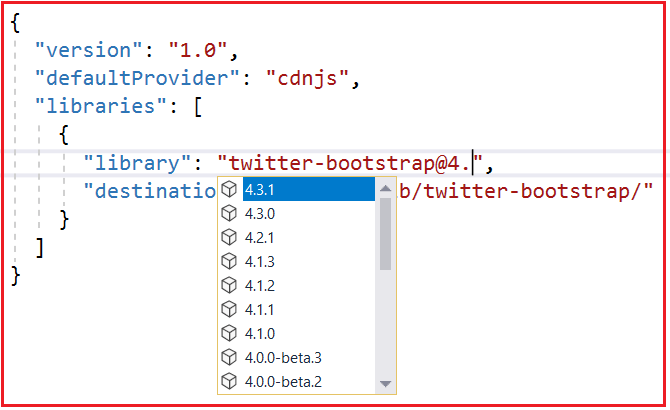
1. Open the libman.json file
2. Click on the client-side library which you want to uninstall or update
3. A light bulb icon will appear at the left side
4. Click on the light bulb icon and then you will see the options whether to update or uninstall that specific client-side library as shown in the below image.



Another approach to uninstall a client-side library is to remove the entry in the libman.json file and upon saving the file the respective client-side libraries are uninstalled from the respective folder location.

Another approach to upgrade or downgrade a client-side library is to change the version number directly in libman.json file and upon saving the file the respective client-side library will be updated to the version you modified.

**Note:** While updating the version number, you will also get the visual studio intelligence as shown in the below image.



**How to Use Bootstrap in ASP.NET Core MVC**

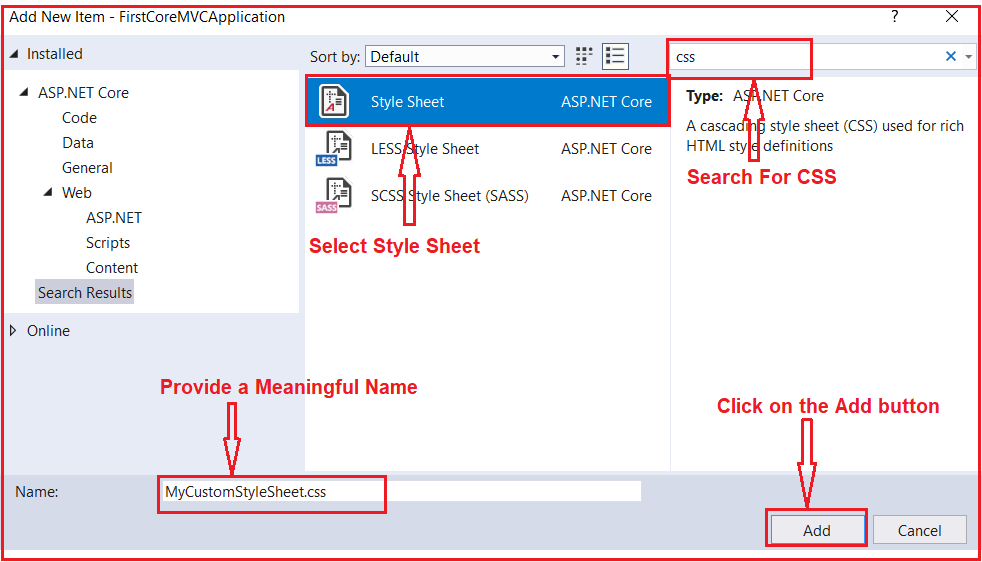
**How to Use Bootstrap in ASP.NET Core MVC Application**

In this article, I am going to discuss **How to Use Bootstrap in ASP.NET Core MVC** Application. Please read our previous article before proceeding to this article where we discussed the [**How to Installed Bootstrap in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/how-to-install-bootstrap-in-asp-net-core/) Application using Library Manager (Libman). Here, I will discuss how to use bootstrap as well as how to create and use custom CSS in a view.

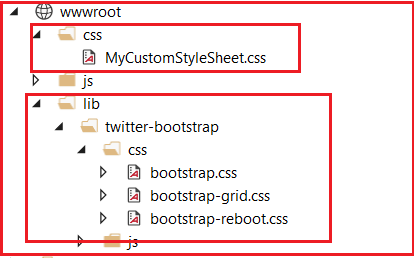
**Creating a Custom Style Sheet in ASP.NET Core MVC Application:**

First, create a folder with the name **CSS** within the **wwwroot** folder. All the custom CSS files are going to be created within this folder. Once you create the CSS folder, let’s add a CSS file with the name **MyCustomStyleSheet.css**.

To create a style sheet, right-click on the CSS folder and then select “**Add – New Item**” from the context menu. Then search for css and select Style Sheet, provide a meaningful name and finally click on the Add button as shown in the below image.



Once you add **MyCustomStyleSheet.css** file, then your wwwroot folder should looks as shown below.



**Note:** All the site custom CSS need to be placed within the **MyCustomStyleSheet.css** file. So, open the MyCustomStyleSheet.css file and copy and paste the following code in it.

**.btn {**  
**width: 80px;**  
**}**

**How to Using Bootstrap in ASP.NET Core MVC Application?**

In order to use bootstrap, first, you need to include a reference to the bootstrap.css file. You can add the reference in each individual views. But as we are going to use the **Layout** file, so we are going to add a reference to the **bootstrap.css** file in the **\_Layout.css** file. Along with **bootstrap.css,** we are also including a reference to our custom style sheet i.e. **MyCustomStyleSheet.css**.

**Modifying the \_Layout.cshtm file:**

Please modify the Layout file which is present in the shared folder as shown below.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**<link** href="~/lib/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" **/>**

**<link** href="~/css/MyCustomStyleSheet.css" rel="stylesheet" **/>**

**</head>**

**<body>**

**<div** class="container"**>**

@RenderBody()

**</div>**

**</body>**

**</html>**

As you can see in the HTML code, we have included references for both **bootstrap.css** as well as **MyCustomStyleSheet.css** files. Here, we are also using the bootstrap container class for positioning the elements on the page.

**Creating Models:**

Within the Models folder, add a class file with the name **Student.cs** and then copy and paste the following code in it.

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

**Modifying the Home Controller:**

Please modify the Home Controller as shown below.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 102, Name = "Smith", Branch = "ETC", Section = "B", Gender = "Male" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 104, Name = "Sara", Branch = "CSE", Section = "A", Gender = "Female" **}**,

new Student**()** **{** StudentId = 105, Name = "Pam", Branch = "ETC", Section = "B", Gender = "Female" **}**

**}**;

**return** View**(**listStudents**)**;

**}**

**public** ViewResult Details**(int** Id**)**

**{**

var studentDetails = new Student**()** **{** StudentId = Id, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**;

**return** View**(**studentDetails**)**;

**}**

**}**

**}**

**Modifying the Startup class:**

Please modify the Startup class as shown below. Here, to serve the bootstrap we need to add the static files middle layer before the MVC middle layer in the request processing pipeline.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.Extensions.DependencyInjection;*

**namespace** *FirstCoreMVCApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddMvc**()**;

**}**

**public** **void** Configure**(**IApplicationBuilder app, IHostingEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

//Adding Static Files Middleware to serve the static files

app.UseStaticFiles**()**;

//Adding MVC Middleware

app.UseMvcWithDefaultRoute**()**;

**}**

**}**

**}**

**Modifying the Index view:**

Please modify the Index view of Home Controller as shown below.

@model List**<FirstCoreMVCApplication.Models.Student>**

@{

ViewBag.Title = "Student List";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="table-responsive"**>**

**<table** class="table"**>**

**<thead>**

**<tr>**

**<th>**ID**</th>**

**<th>**Name**</th>**

**<th>**View**</th>**

**<th>**Update**</th>**

**<th>**Delete**</th>**

**</tr>**

**</thead>**

**<tbody>**

@foreach (var student in Model)

{

**<tr>**

**<td>**@student.StudentId**</td>**

**<td>**@student.Name**</td>**

**<td** class="text-center"**><a** href="#" class="btn btn-primary"**>**View**</a></td>**

**<td** class="text-center"**><a** href="#" class="btn btn-primary"**>**Edit**</a></td>**

**<td** class="text-center"**><a** href="#" class="btn btn-danger"**>**Delete**</a></td>**

**</tr>**

}

**</tbody>**

**</table>**

**</div>**

**Modifying the Details View:**

Please modify the Details view as shown below.

@model FirstCoreMVCApplication.Models.Student

@{

ViewBag.Title = "Student Details";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="row justify-content-center m-3"**>**

**<div** class="col-sm-8"**>**

**<div** class="card"**>**

**<div** class="card-header text-center"**>**

**<h1>**@Model.Name**</h1>**

**</div>**

**<div** class="card-body text-center"**>**

**<h4>**Studnet ID : @Model.StudentId**</h4>**

**<h4>**Branch : @Model.Branch**</h4>**

**<h4>**Section : @Model.Section**</h4>**

**<h4>**Gender : @Model.Gender**</h4>**

**</div>**

**<div** class="card-footer text-center"**>**

**<a** href="#" class="btn btn-primary"**>**Back**</a>**

**<a** href="#" class="btn btn-primary"**>**Edit**</a>**

**<a** href="#" class="btn btn-danger"**>**Delete**</a>**

**</div>**

**</div>**

**</div>**

**</div>**

That’s it. Save the changes and run the application and see the output as expected. Here, we have just created the View, Update, Delete, Back buttons but not implemented. In our upcoming articles, I will show you how to implement the CRUD operation in ASP.NET Core MVC Application.

**Tag Helpers in ASP.NET Core MVC**

**Tag Helpers in ASP.NET Core MVC**

In this article, I am going to discuss **Tag Helpers in ASP.NET Core MVC Application** with some examples. Tag helpers are one of the new features introduced in ASP.NET Core MVC**.** Please read our previous article, where we discussed [**how to use bootstrap in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/how-to-use-bootstrapin-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers.

1. **What are Tag Helpers in ASP.NET Core?**
2. **Types of Tag Helpers**
3. **How to use built-in Tag Helpers in ASP.NET Core MVC?**
4. **Understanding the Anchor Tag Helper in ASP.NET Core**
5. **Advantage of using Tag helpers in ASP.NET Core MVC Application**

**What are Tag Helpers in ASP.NET Core?**

Tag Helpers in ASP.NET Core are the server-side components. They are basically used to perform defined transformations on HTML Elements. As they are server-side components, so they are going to be processed on the server to create and render HTML elements in the Razor files.

If you are coming from ASP.NET MVC background, then you may be worked with the HTML helpers. The Tag Helpers are similar to the HTML helpers.

**Types of Tag Helpers in ASP.NET Core:**

There are two types of Tag helpers in ASP.NET Core. They are as follows:

1. **Built-In Tag Helpers**: They come in-built in the ASP.NET Core Framework and can perform common tasks like generating links, creating forms, loading assets, showing validation messages, etc.
2. **Custom Tag Helper**: That can be created by us to perform our desired transformation on an HTML element.

**Note:** In this article, I am going to discuss the Built-in Tag Helpers and in our upcoming articles, I will discuss the Custom Tag Helpers.

Please read the following articles before proceeding to this article as we are going to use layout, ViewImport, and bootstrap in this demo.

[**How to create and use Layout in ASP.NET Core?**](https://dotnettutorials.net/lesson/layout-view-in-asp-net-core-mvc/)  
[**Understanding the need and use of \_ViewImport.cshtml file.**](https://dotnettutorials.net/lesson/viewimports-in-asp-net-core-mvc/)  
[**How to install bootstrap in ASP.NET Core?**](https://dotnettutorials.net/lesson/how-to-install-bootstrap-in-asp-net-core/)  
[**Using Bootstrap in ASP.NET Core.**](https://dotnettutorials.net/lesson/how-to-use-bootstrapin-asp-net-core-mvc/)

**How to use built-in Tag Helpers in ASP.NET Core MVC?**

In order to make the built-in tag helpers available for all the views of our application, import the tag helpers using the **\_ViewImports.cshtml** file. You need to import them using the **@addTagHelper**directive as shown below.

**@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers**

The **@addTagHelper** makes the built-in tag helpers to available in the application which are defined in an assembly called **Microsoft.AspNetCore.Mvc.TagHelpers**. Here the wildcard **“\*”** specifies that all the Tag Helpers are made available.

**Generating Links using Tag Helpers in ASP.NET Core MVC Application:**

Let us understand this with an example. First create a class file with the name Student.cs within the Models folder. Once you create the Student.cs class file, then copy and paste the following code in it.

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** string Section **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

**Modifying Home Controller:**

Modify the Home Controller as shown below. Here we have created two action methods. To make the things simple and to keep focus on Tag helpers, here we have hardcode the required student data within the action method.

**using** *FirstCoreMVCApplication.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 102, Name = "Smith", Branch = "ETC", Section = "B", Gender = "Male" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = "CSE", Section = "A", Gender = "Male" **}**,

new Student**()** **{** StudentId = 104, Name = "Sara", Branch = "CSE", Section = "A", Gender = "Female" **}**,

new Student**()** **{** StudentId = 105, Name = "Pam", Branch = "ETC", Section = "B", Gender = "Female" **}**

**}**;

**return** View**(**listStudents**)**;

**}**

**public** ViewResult Details**(int** Id**)**

**{**

var studentDetails = new Student**()** **{** StudentId = Id, Name = "James", Branch = "CSE", Section = "A", Gender = "Male" **}**;

**return** View**(**studentDetails**)**;

**}**

**}**

**}**

**Creating the Details view:**

**Details.cshtml:**

@model FirstCoreMVCApplication.Models.Student

@{

ViewBag.Title = "Student Details";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="row justify-content-center m-3"**>**

**<div** class="col-sm-8"**>**

**<div** class="card"**>**

**<div** class="card-header text-center"**>**

**<h1>**@Model.Name**</h1>**

**</div>**

**<div** class="card-body text-center"**>**

**<h4>**Studnet ID : @Model.StudentId**</h4>**

**<h4>**Branch : @Model.Branch**</h4>**

**<h4>**Section : @Model.Section**</h4>**

**<h4>**Gender : @Model.Gender**</h4>**

**</div>**

**<div** class="card-footer text-center"**>**

**<a** href="#" class="btn btn-primary"**>**Back**</a>**

**</div>**

**</div>**

**</div>**

**</div>**

**Index View:**

Now in the index view, we have to provide the View button as a link and on the click, on the View button, we need to display the Student Details. For example, we need to generate the following hyperlink. The number 101 is the ID of the student whose details we want to view.

**/home/details/101**

We have many different options to generate a link in ASP.NET Core MVC Application. Let us discuss all the possible options to generate a link and then we will discuss why we should use Tag Helper over others.

**Option1: Manually generating the links:**

@model List**<FirstCoreMVCApplication.Models.Student>**

@{

ViewBag.Title = "Student List";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="table-responsive"**>**

**<table** class="table"**>**

**<thead>**

**<tr>**

**<th>**ID**</th>**

**<th>**Name**</th>**

**<th>**View**</th>**

**</tr>**

**</thead>**

**<tbody>**

@foreach (var student in Model)

{

**<tr>**

**<td>**@student.StudentId**</td>**

**<td>**@student.Name**</td>**

**<td** class="text-center"**><a** href="/home/details/@student.StudentId"**>**View**</a></td>**

**</tr>**

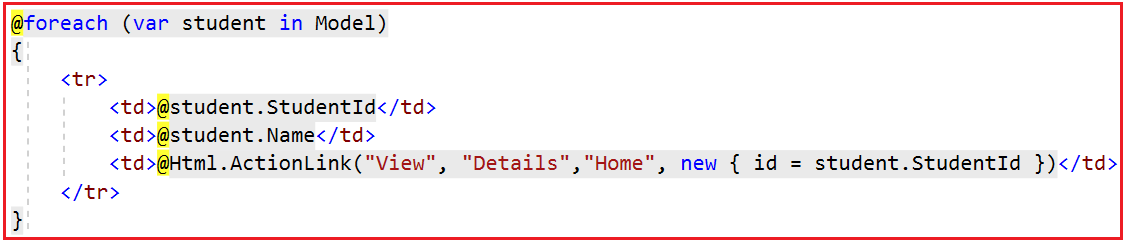
}

**</tbody>**

**</table>**

**</div>**

**Option2: Using HTML helpers**



**Option3: Using Tag Helpers:**

In order to use Tag Helpers first import the **@addTagHelper** directive in the **\_ViewImport.cshtml** file. Along with the **@addTagHelper** directive, we also add the model namespace using the **@using** directive. So, modify the **\_ViewImport.cshtml** file as shown below which you can find within the Views folder.

@using FirstCoreMVCApplication.Models;

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

**Then modify the Index view as shown below.**

@model List**<Student>**

@{

ViewBag.Title = "Student List";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="table-responsive"**>**

**<table** class="table"**>**

**<thead>**

**<tr>**

**<th>**ID**</th>**

**<th>**Name**</th>**

**<th>**View**</th>**

**</tr>**

**</thead>**

**<tbody>**

@foreach (var student in Model)

{

**<tr>**

**<td>**@student.StudentId**</td>**

**<td>**@student.Name**</td>**

**<td>**

**<a** asp-controller="Home" asp-action="Details"

asp-route-id="@student.StudentId"**>**View**</a>**

**</td>**

**</tr>**

}

**</tbody>**

**</table>**

**</div>**

Now run the application and it should work as expected.

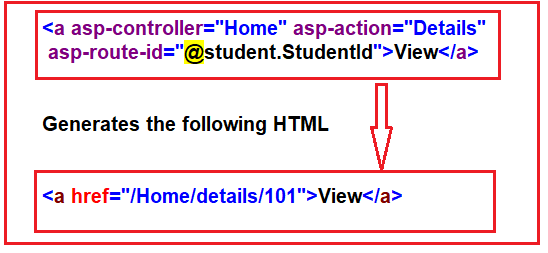
**Understanding the Anchor Tag Helper in ASP.NET Core:**

The Anchor Tag Helper in ASP.NET Core creates the standard HTML anchor **(<a … ></a>)** tag by adding new attributes such as:

1. **asp-controller:**It is used to specify the controller to target based on the routing system. If you omitted this, then the controller of the current view is used by default.
2. **asp-action:**It is used to specify the Action method to target based on the routing system. If you omitted this attribute then the action rendering the current view is used by default.
3. **asp-route-{value}:**It is used for specifying the additional segment value for the URL. For example, asp-route-id is used to provide value for the ‘id’ segment.

The rendered anchor element’s “href” attribute value is determined by the values of these “**asp-“** attributes. As the name says, **asp-controller** specifies the name of the controller whereas **asp-action** specifies the name of the action name. Similarly, the **asp-route-{value}** attribute is used to include route data in the generated href attribute value. **{value}** can be replaced with route parameters such as id, name, etc.

**Let us have a look at the following image.**



As you can see in the above image, manually generating the links is much easier than using HTML Helpers or Tag Helpers. Then why should we use **HTML helpers** or **Tag Helpers**over manually generating these links in ASP.NET Core?

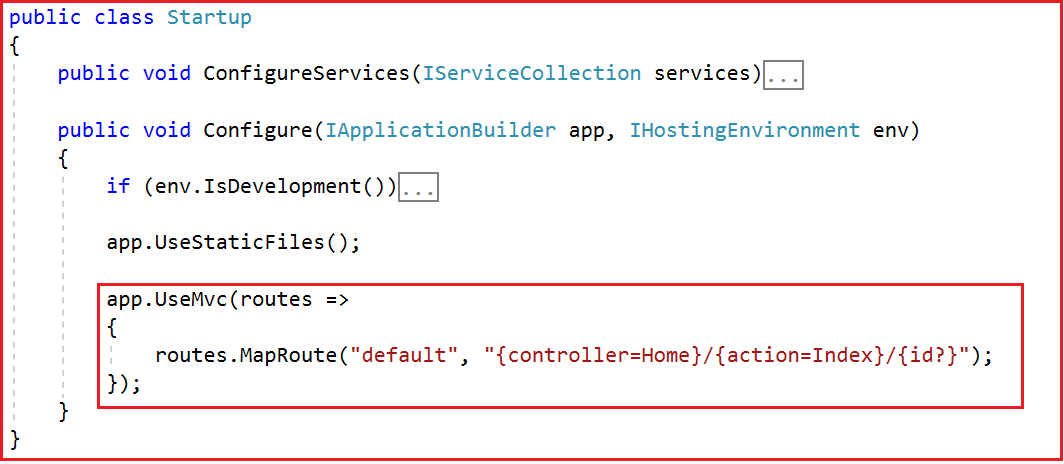
**Advantage of using Tag helpers in ASP.NET Core MVC Application:**

In ASP.NET Core MVC, the Tag Helpers generates link based on the application routing templates. That means. in future, if we change routing templates, then the link generated by tag helpers will automatically reflect those changes made to the routing templates. So, the generated links just work as expected without any trouble.

On the other hand, if we have hard-coded the URL paths manually, then we need to change the code wherever we have hardcoded the path in our application when the application routing templates change.

**Let’s understand this with an example**

The following is the Startup class of our application. Please modify the Startup class as shown below.



As you can see in the above image, we have the route template pattern as **{controller=Home}/{action=Index}/{id?}.**

**Generating Link Manually:**

In the following code, we are manually generating the link by hard-coding the URL paths as /Home/Details.

**<a href=”/Home/Details/@student.StudentId”>View</a>**

**Generating Link using Tag Helper:**

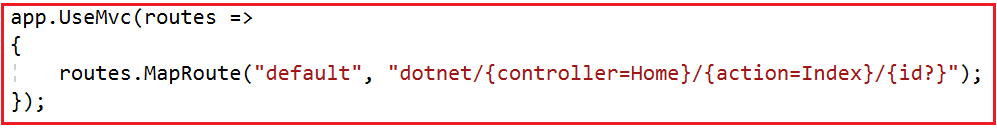
In the following code, we are generating the link using anchor tag helper.

**<a asp-controller=”Home” asp-action=”Details” asp-route-id=”@student.StudentId”>View</a>**

As you can see with Tag Helper, we have not hardcoded the URL paths. Here, we are only specifying the controller and action names and the route parameters along with their values. When the tag helpers are executed on the server they look at the route templates and generate the correct URLs automatically.

Here, in both the techniques generate the same URL path i.e. (**/Home/Details/101**) and it also works with the current route template i.e. (**{controller=Home}/{action=Index}/{id?}**)

Now let us change the routing template as shown below. Notice here in the route template, we have added the string literal “dotnet”.



With the above changes in place, now if you generate the link manually, then it will not work whereas if you generate the link with Tag Helpers then it will work as expected.

We also have tag helpers in ASP.NET Core to generate forms. When the form is posted back to the server, the posted form values are automatically handled and the associated validation messages are displayed. Without these tag helpers, we would have to write a lot of custom code to achieve the same. If this is not clear at the moment then don’t worry we will discuss this with examples in our upcoming articles.

**Image Tag Helper in ASP.NET Core**

**Image Tag Helper in ASP.NET Core MVC Application**

In this article, I am going to discuss the **Image Tag Helper in ASP.NET Core MVC** Application with some examples. Please read our previous article, where we discussed the basics of [**Tag Helpers in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/tag-helpers-in-asp-net-core-mvc/) Application. As part of this article, we are going to discuss the following pointers. As part of this article, we are going to discuss the following pointers.

1. **Understanding the Browser Cache**
2. **How to Disable Browser Cache?**
3. **Why we need Image Tag Helper in ASP.NET Core?**
4. **How to use Image Tag Helper in ASP.NET Core?**
5. **Understanding ASP.NET Core Image Tag Helper with an example**
6. **How Does the Image Tag Helper work in ASP.NET Core?**

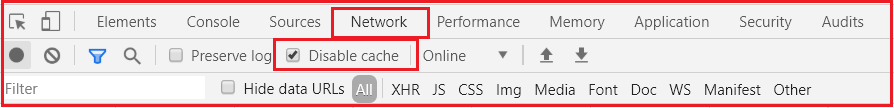
**Understanding the Browser Cache:**

In general, when you visit a web page and if that web page contains some image, then most of the modern web browser cache the images for later use. In the future, when you revisit to that web page, then the browser loads the images from cache instead of downloading the images from the server. In most of the caches, this is not an issue as the images are not changes that often for a web page.

**How to Disable Browser Cache?**

If you want then you can also disable the browser cache. Once you disable the browser cache then it will not cache the images and each and every time it will download the images from the server when you visit the page. In order to disable browser cache in Google Chrome, you need to follow below steps.

Press **F12** key to launch the Browser Developer Tools. Then click on the **“Network”** tab and finally, check then **“Disable Cache”** checkbox as shown in the below image.



**Why we need Image Tag Helper in ASP.NET Core?**

If you disable the browser cache, then each and every time it will download the image from the server. If you didn’t disable the browser cache, then it will serve the image from the browser cache. But the problem with this approach is that, if you changed the image, then you will not get the updated image.

To overcome the above problems, i.e. download the image from the server only when it has changed else serve the image from the browser cache, we need to use the Image Tag helper in ASP.NET Core MVC Application.

**How to use Image Tag Helper in ASP.NET Core?**

In order to use the Image Tag Helper in ASP.NET Core Application, you need to add the **asp-append-version** attribute to the <**img>** tag and need to set the value to true as shown below.

How to use Image Tag Helper in ASP.NET Core

**Understanding ASP.NET Core Image Tag Helper with an example:**

First, create a folder with the name **images** within the **wwwroot** folder. Once you create the **images** folder, add two images with the name Image1.jpg and Image2.jpg. For better understanding add two different images.

**Modify the Home Controller as shown below.**

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**()**;

**}**

**}**

**}**

**Modify the Index View as shown below.**

@{

ViewBag.Title = "Index";

Layout = null;

}

**<img** src="~/images/Image1.jpg" asp-append-version="true" **/>**

**Modify the \_ViewImports.cshtml file as shown below.**

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

With the above changes in place, now run the application and view the page source code. Once you view the page source code, you will find the following code for the image tag.

How Does the Image Tag Helper work in ASP.NET Core

**How Does the Image Tag Helper work in ASP.NET Core?**

The ASP.NET Core **Image Tag Helper** enhances the **<img>** tag to provide **cache-busting**behavior for the static image files. That means based on the content of the image, a unique hash value is calculated and then appended to image URL as you can see in the above image. This unique hash value decides whether to load the image from the server or to load the image from the browser cache. If the hash value changes then it will reload the image from the server else it will reload the image from the cache.

Now reload the page multiple times and you will see the hash is not going to change and hence it will load the image from the browser cache.

Now, rename Image1 as Image3 and Image2 as Image1 within the images folder and reload the page and you will see a different hash value as well as the updated image which proofs that it loads the image from the server.

# Environment Tag Helper in ASP.NET Core

## ****Environment Tag Helper in ASP.NET Core Application****

In this article, I am going to discuss the **Environment Tag Helper in ASP.NET Core**Application with some examples. Please read our previous article, where we discussed the [**Image Tag Helper in ASP.NET Core**](https://dotnettutorials.net/lesson/image-tag-helper-asp-net-core/)Application with an example. As part of this article, we are going to discuss the following pointers.

**1. Different Environments in Software Development**  
**Development Environment**  
**Staging Environment**  
**Production Environment**  
**2. Understanding the use of Environment Tag Helper in ASP.NET Core**  
**3. How to set Environment in ASP.NET Core Application?**  
**4. What is the use of integrity in CDN Link?**  
**5. What if the CDN is down?**  
**6. Multiple Examples to understand the ASP.NET Core Environment Tag Helper**

##### ****Different Environments in Software Development:****

Nowadays, most of the software development organizations typically have the following three development environments.

1. **Development Environment**
2. **Staging Environment**
3. **Production Environment**

##### ****Development Environment:****

As a software developer, for our day to day development work, we generally use this development environment. In this environment, we generally use the non-minified JavaScript and CSS files for the purpose of easy debugging. Another use of this environment is that we want to show the developer exception page if there is an unhandled exception so that we can understand the root cause of the exception and then take necessary action to fix the issue.

##### ****Staging Environment:****

The staging environment is very much similar to the production environment. Nowadays, many organizations use this staging environment to identify any deployment-related issues. Again, if you are developing a B2B (Business to Business) application, then you may be using services provided by other service providers. So, many organizations set up their staging environment to check the service providers as well for complete end to end testing.

In the staging environment, we usually do not perform the debugging and troubleshoot, so to get a better performance we use minified JavaScript and CSS files. Again, if there is an exception, instead of showing the developer exception page, we need to show a friendly error page. The friendly error page will not contain any technical exception details instead it shows a generic error message like below.

**“Something went wrong, our IT team working on this to solve as soon as possible. If you need further details then please email, chat or call our support team using the below contact details”**

##### ****Production Environment:****

This is the actual live environment which is used for day to day business. The Production environment should be configured for maximum security as well as maximum performance. So, in this environment, we need to use the minified JavaScript and CSS files. For better security, we need to show a User-Friendly Error Page instead of the Developer Exception Page when there is an exception.

##### ****Understanding the use of Environment Tag Helper in ASP.NET Core:****

In our Application, we are using bootstrap. For ease of debugging, i.e. on our local development (i.e. on Development Environment) we want to load the non-minified bootstrap CSS file (bootstrap.css).

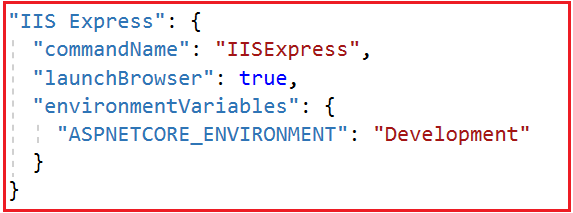
On the Staging and Production or any other environment except the Development environment, we want to load the minified bootstrap CSS file (bootstrap.min.css) from a CDN (Content Delivery Network) for better performance.

However, if the CDN is down or for some reason, our application is not able to load the file from CDN then we want our application to fallback and load the minified bootstrap file (bootstrap.min.css) from our own server.

We can achieve this very easily by using the **Environment Tag Helper in ASP.NET Core** Application. But before understanding how to use the **Environment Tag Helper**, let us first understand how we set the environment (i.e. Development, Staging, and Production) in ASP.NET Core.

##### ****How to set Environment in ASP.NET Core Application?****

In order to set the Application Environment (i.e. Development, Staging, or Production) we need to use the **ASPNETCORE\_ENVIRONMENT** variable. We can set this in the **launchsettings.json** file. You can find the **launchsettings.json** file within the properties. Open the **launchsettings.json** file and set the environment as Development as shown in the below image.



**Note:** The other possible values for **ASPNETCORE\_ENVIRONMENT**variable are **Staging** and **Production**. It is also possible to create our own environment that we will discuss in our upcoming article.

##### ****Installing bootstrap:****

Please install bootstrap with the **wwwroot** folder. Please read the following article where we discussed how to install bootstrap in ASP.NET Core Application.

<https://dotnettutorials.net/lesson/how-to-install-bootstrap-in-asp-net-core/>

##### ****Modifying the \_ViewImports.cshtml file:****

Please modify the **\_ViewImports.cshtml** file as shown below to include the tag helpers into our application globally.

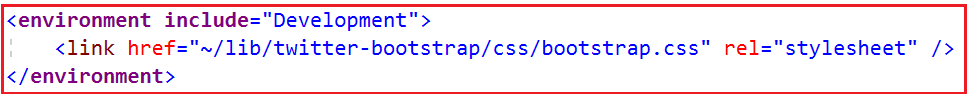
**@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers**

##### ****ASP.NET Core Environment Tag Helper with Examples****:

The **Environment Tag Helper in ASP.NET Core** supports rendering different content based on the environment of the application. The environment of the application is set by using the **ASPNETCORE\_ENVIRONMENT** variable within the **launchsettings.json** file that we already discussed.

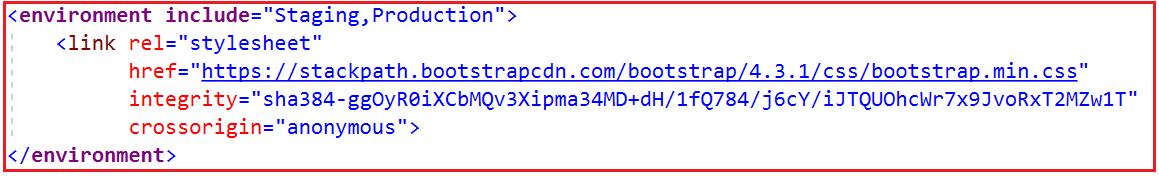
###### **Example:**

If you want to load the non-minified bootstrap CSS files from your own server when the application environment is Development, then you need to set the environment tag helper as shown below.



###### **Example:**

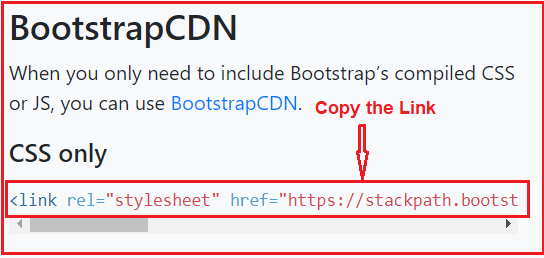
If you want to load the minified bootstrap CSS files from the CDN (Content Delivery Network) when the application environment is Staging or Production, then you need to set the environment tag helper as shown below.



In order to get the bootstrap CDN link, please visit the following URL.

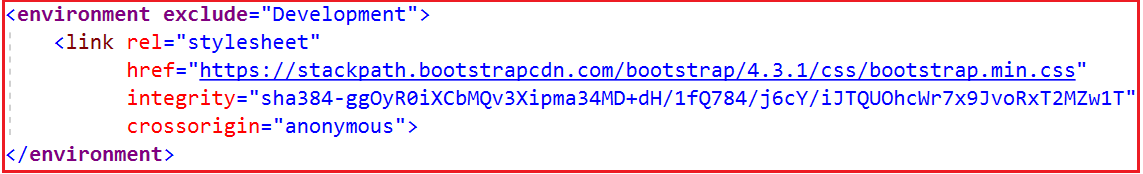
[**https://getbootstrap.com/**](https://getbootstrap.com/)

Once you visit the above URL then copy the CDN link as shown below.



Using the **include** attribute of the environment tag helper we can set a single hosting environment name or a comma-separated list of hosting environment names. Along with the include attribute, we can also use the exclude attribute.

The exclude attribute is basically used to render the content of the environment tag when the hosting environment does not match the environment specified in the exclude the attribute. The following example loads the minified CSS file from the CDN when the application environment is not Development.

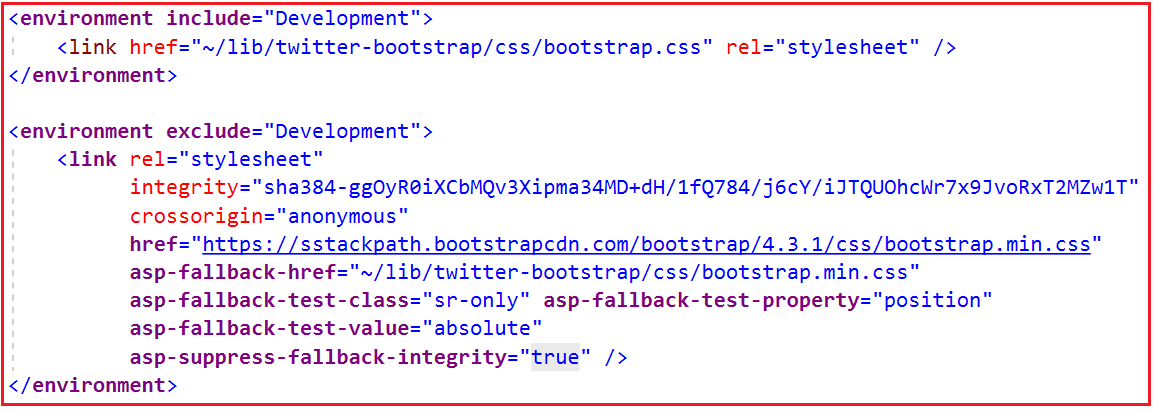


##### ****What is the use of integrity in CDN Link?****

The **integrity** attribute on the <link> element is used for **Subresource Integrity (SRI) check**. The SRI is a security feature which allows a browser to check if the file is retrieved from the CDN has been maliciously altered. When the browser downloads the file, it regenerates the hash and compares the regenerate the hash value against the integrity attribute hash value. If both the hash values match, then only the browser allows the file to be downloaded otherwise it is blocked.

##### ****What if the CDN is down?****

If the CDN is down or for some reason, our application is not able to reach the CDN, then we want our application to fall back and load the minified bootstrap file (bootstrap.min.css) from our own server. Please have a look at the following image.



##### ****Explanation of the above Code:****

If the environment is “Development”, then we are loading the non-minified bootstrap CSS file (i.e. bootstrap.css) from our own server. On the other hand, if the application environment is other than “Development”, then we are trying the load the minified bootstrap CSS file (i.e. bootstrap.min.css) from the CDN.

Along the same line, we also specified a fallback source using the **asp-fallback-href** attribute. This attribute is basically used when the CDN is down or for some reason if our application is unable to reach the CDN. In that case, our application falls back and download the minified bootstrap file (i.e. bootstrap.min.css) from our own server.

In order to check whether the CDN is down, we are using the following 3 attributes and their associated values.

1. **asp-fallback-test-class=”sr-only”**
2. **asp-fallback-test-property=”position”**
3. **asp-fallback-test-value=”absolute”**

Let us understand all the above with an example.

##### ****Modifying the \_Layout.cshtml file:****

Please modify the **\_Layout.cshtml** file as shown below. Here, we are loading the bootstrap.css files based on the application environment. We also specify the falls back path in case if the environment is other than Development and the CDN is down.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**<environment** include="Development"**>**

**<link** href="~/lib/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" **/>**

**</environment>**

**<environment** exclude="Development"**>**

**<link** rel="stylesheet"

integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"

crossorigin="anonymous"

href="https://sstackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"

asp-fallback-href="~/lib/twitter-bootstrap/css/bootstrap.min.css"

asp-fallback-test-class="sr-only" asp-fallback-test-property="position"

asp-fallback-test-value="absolute"

asp-suppress-fallback-integrity="true" **/>**

**</environment>**

**</head>**

**<body>**

**<div** class="container"**>**

@RenderBody()

**</div>**

**</body>**

**</html>**

##### ****Modify the Home Controller:****

Please modify the Home Controller as shown below.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ViewResult Index**()**

**{**

**return** View**()**;

**}**

**}**

**}**

##### ****Modify the Index.cshtml file as shown below.****

@{

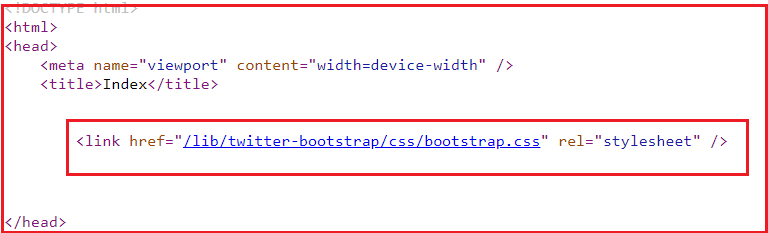
ViewBag.Title = "Index";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<h1>**Index View**</h1>**

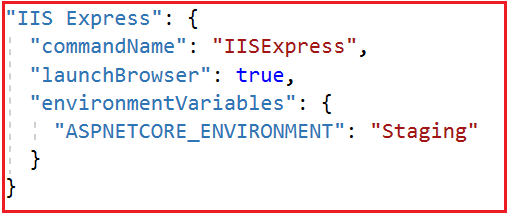
**Now run the application and view the page source code.**



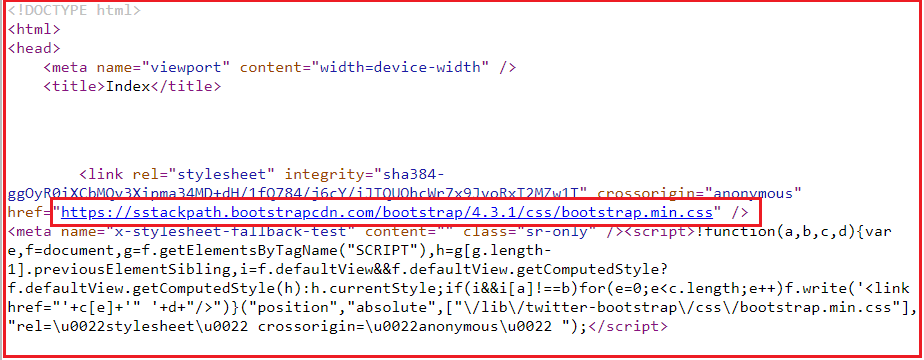
As you can see in the above image, the boostrap.css file is load from our own server and this makes sense because currently, we set the application environment as Development. Let us change the environment to Staging and see whether it load the file from CDN or not.

##### ****Modify the Environment as Staging:****

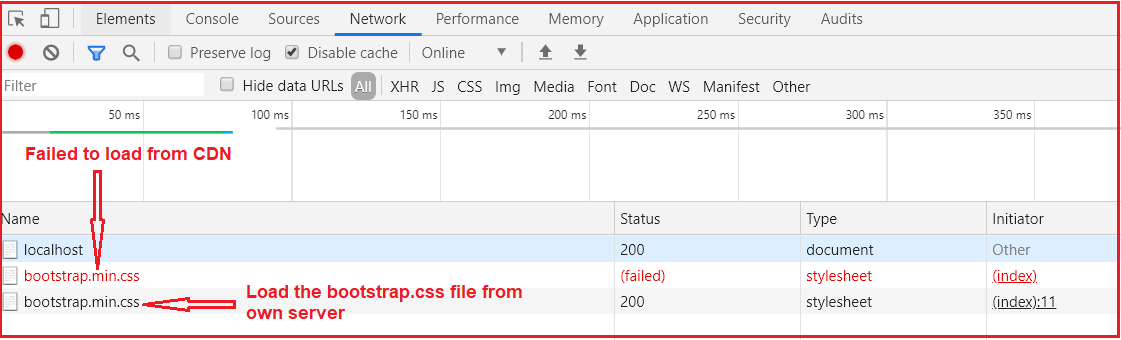
Open the **launchsettings.json** file and modify the **ASPNETCORE\_ENVIRONMENT** variable value to staging as shown below.



With the above changes in place, now run the application and view the source, which should show the bootsrap.css file being downloaded from the CDN as shown below.



Let intentionally change the integrity value of the CDN to make sure if the CDN fails whether it load the minified bootstrap.css file from our local server or not. So, add some random value to the value of integrity attribute of CDN link. And then load the page and view the page source code. Now, in order to check this open the browser developer tools by pressing the F12 key and then click on the Network tab as shown below.



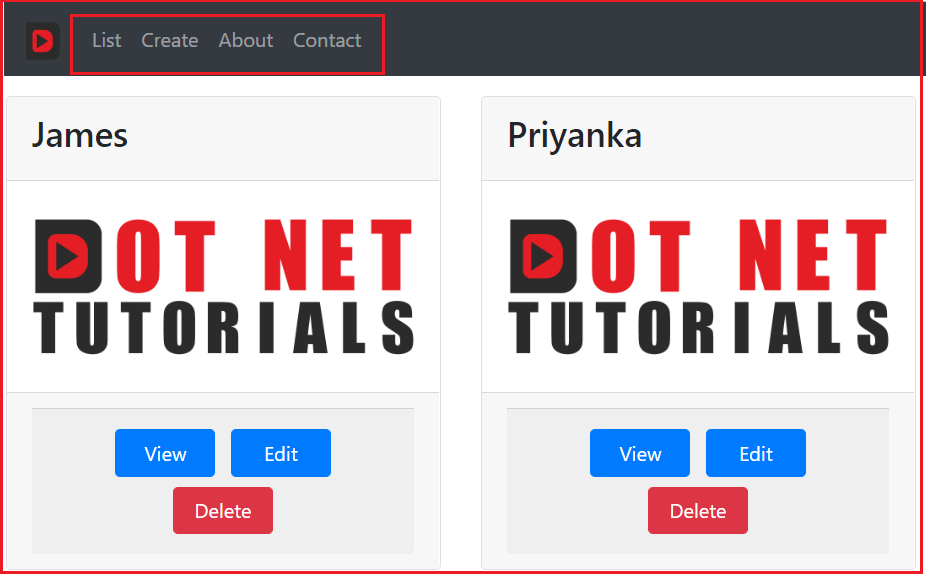
As you can see in the above image, first it tries to load the bootstrap.css file from CDN and it fails as the integrity value is not matched. Then it loads the file from own application server.

**Navigation Menus in ASP.NET Core**

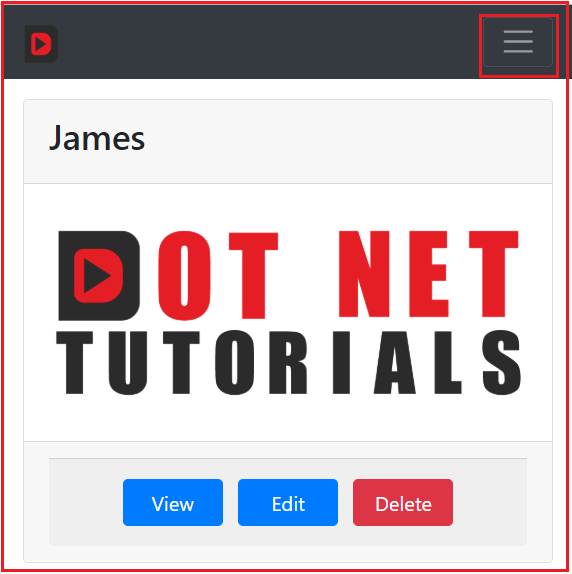
**Navigation-Menus in ASP.NET Core Application**

In this article, I am going to discuss **how to create Responsive Navigation Menus in ASP.NET Core Application** using bootstrap and JQuery. Please read our previous article, where we discussed the [**Environment Tag Helper in ASP.NET Core**](https://dotnettutorials.net/lesson/environment-tag-helper-asp-net-core/) Application.

On a large screen device, we want the navigation menus to look as shown below.



On a small screen device, we want to show the navigation menus like below.



**Adding bootstrap and JQuery files:**

The most important point that you need to remember is Bootstrap 4 has a dependency on jQuery. So, here we need to download both bootstraps as well as JQuery into our application. Here, we are going to use a tool called Library Manager (LibMan) to download the required bootstrap and JQuery files. If you are new to Library Manager then I strongly recommended you to read the following article where we discussed how to use LibMan to download client-side libraries.

[**https://dotnettutorials.net/lesson/how-to-install-bootstrap-in-asp-net-core/**](https://dotnettutorials.net/lesson/how-to-install-bootstrap-in-asp-net-core/)

**Adding images Folder:**

Add a folder called images with the wwwroot folder and then paste two different images with the name **Logo.png** and **Student.png**.

**Adding css Folder:**

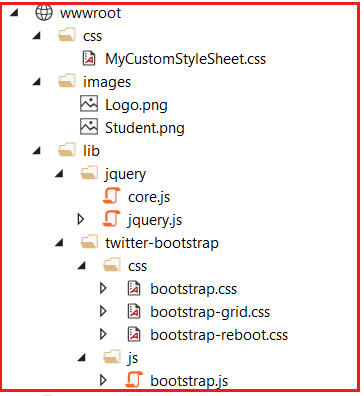
Again add a folder with the name css within the wwwroot folder and then add a css file with the name **MyCustomStyleSheet.css**. Once you create the css file, then copy and paste the following code in it.

**.btn** **{**

width: 80px;

**}**

With the above files and folders in place, your **wwwroot** folder should looks as shown below.



**\_ViewImports.cshtml file:**

Please modify the **\_ViewImports.cshtml** file as shown below.

@using FirstCoreMVCApplication.Models

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

**\_Layout.cshtml file:**

Please modify the **\_Layout.cshtml** file as shown below.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**@ViewBag.Title**</title>**

**<link** href="~/css/MyCustomStyleSheet.css" rel="stylesheet" **/>**

**<link** href="~/lib/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" **/>**

**<script** src="~/lib/jquery/jquery.js"**></script>**

**<script** src="~/lib/twitter-bootstrap/js/bootstrap.js"**></script>**

**</head>**

**<body>**

**<div** class="container"**>**

**<nav** class="navbar navbar-expand-sm bg-dark navbar-dark"**>**

**<a** class="navbar-brand" asp-controller="home" asp-action="index"**>**

**<img** src="~/images/Logo.png" width="30" height="30"**>**

**</a>**

**<button** class="navbar-toggler" type="button" data-toggle="collapse" data-target="#collapsibleNavbar"**>**

**<span** class="navbar-toggler-icon"**></span>**

**</button>**

**<div** class="collapse navbar-collapse" id="collapsibleNavbar"**>**

**<ul** class="navbar-nav"**>**

**<li** class="nav-item"**>**

**<a** class="nav-link" asp-controller="home" asp-action="index"**>**List**</a>**

**</li>**

**<li** class="nav-item"**>**

**<a** class="nav-link" asp-controller="home" asp-action="create"**>**Create**</a>**

**</li>**

**<li>**

**<a** class="nav-link" asp-controller="home" asp-action="about"**>**About**</a>**

**</li>**

**<li>**

**<a** class="nav-link" asp-controller="home" asp-action="contact"**>**Contact**</a>**

**</li>**

**</ul>**

**</div>**

**</nav>**

**<div>**

@RenderBody()

**</div>**

**</div>**

**</body>**

**</html>**

**Note:** On a small screen device, for the **navbar**toggle button to work, the jQuery reference must be loaded before loading the Bootstrap JavaScript file. If you change the order, then the **navbar toggle button**does not work as expected.

**Creating Models:**

First, we need to create two enums to store the Gender and Branch of a student. So, create two models within the Models folder with the name Gender and Branch and then copy and paste the following code.

**Branch.cs**

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** enum Branch

**{**

None,

CSE,

ETC,

Mech

**}**

**}**

**Gender.cs**

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** enum Gender

**{**

Male,

Female

**}**

**}**

Please add the following Student model with the Models folder.

**Student.cs**

**using** *System.Collections.Generic;*

**namespace** *FirstCoreMVCApplication.Models*

**{**

**public** **class** Student

**{**

**public** **int** StudentId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Email **{** **get**; **set**; **}**

**public** Branch Branch **{** **get**; **set**; **}**

**public** Gender Gender **{** **get**; **set**; **}**

**public** string Address **{** **get**; **set**; **}**

**public** IEnumerable**<**Gender**>** AllGenders **{** **set**; **get**; **}**

**}**

**}**

**Modifying the Home Controller:**

Please modify the Home Controller as shown below.

**using** *FirstCoreMVCApplication.Models;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FirstCoreMVCApplication.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**private** List**<**Student**>** listStudents = new List**<**Student**>()**;

**public** HomeController**()**

**{**

listStudents = new List**<**Student**>()**

**{**

new Student**()** **{** StudentId = 101, Name = "James", Branch = Branch.CSE, Gender = Gender.Male, Address = "A1-2018", Email = "James@g.com" **}**,

new Student**()** **{** StudentId = 102, Name = "Priyanka", Branch = Branch.ETC, Gender = Gender.Female, Address = "A1-2019", Email = "Priyanka@g.com" **}**,

new Student**()** **{** StudentId = 103, Name = "David", Branch = Branch.CSE, Gender = Gender.Male, Address = "A1-2020", Email = "David@g.com" **}**

**}**;

**}**

**public** ViewResult Index**()**

**{**

**return** View**(**listStudents**)**;

**}**

**public** ViewResult Details**(int** Id**)**

**{**

var studentDetails = listStudents.FirstOrDefault**(**std =**>** std.StudentId == Id**)**;

**return** View**(**studentDetails**)**;

**}**

**}**

**}**

**Index.cshtml file:**

Please modify the Index view as shown below.

@model List**<Student>**

@{

ViewBag.Title = "Student List";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="card-deck"**>**

@foreach (var student in Model)

{

**<div** class="card m-3"**>**

**<div** class="card-header"**>**

**<h3>**@student.Name**</h3>**

**</div>**

**<img** class="card-img-top" src="~/images/Student.png" **/>**

**<div** class="card-footer text-center"**>**

**<div** class="card-footer text-center"**>**

**<a** asp-controller="home" asp-action="details"

asp-route-id="@student.StudentId" class="btn btn-primary m-1"**>**View**</a>**

**<a** href="#" class="btn btn-primary m-1"**>**Edit**</a>**

**<a** href="#" class="btn btn-danger m-1"**>**Delete**</a>**

**</div>**

**</div>**

**</div>**

}

**</div>**

**Detail.cshtml file:**

Please modify the Details view as shown below.

@model Student

@{

ViewBag.Title = "Student Details";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<div** class="row justify-content-center m-3"**>**

**<div** class="col-sm-8"**>**

**<div** class="card"**>**

**<div** class="card-header"**>**

**<h1>**@Model.Name**</h1>**

**</div>**

**<div** class="card-body text-center"**>**

**<img** class="card-img-top" src="~/images/Student.png" **/>**

**<h4>**Studnet ID : @Model.StudentId**</h4>**

**<h4>**Email : @Model.Email**</h4>**

**<h4>**Branch : @Model.Branch**</h4>**

**<h4>**Gender : @Model.Gender**</h4>**

**<h4>**Address : @Model.Address**</h4>**

**</div>**

**<div** class="card-footer text-center"**>**

**<a** href="#" class="btn btn-primary"**>**Back**</a>**

**<a** href="#" class="btn btn-primary"**>**Edit**</a>**

**<a** href="#" class="btn btn-danger"**>**Delete**</a>**

**</div>**

**</div>**

**</div>**

**</div>**

**Form Tag Helpers in ASP.NET Core**

**Form Tag Helpers in ASP.NET Core**

In this article, I am going to discuss the **Form Tag Helpers in ASP.NET Core**Application. Please read our previous article where we discussed how to create [**Responsive Navigation Menus in ASP.NET Core**](https://dotnettutorials.net/lesson/navigation-menus-asp-net-core/) Application.

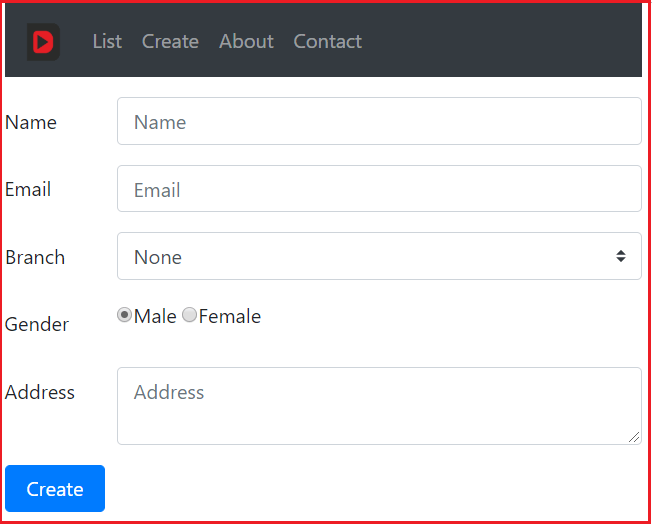
As of now, we have discussed the use of anchor, image and environment tag helpers in ASP.NET Core Application. This is a continuation part of our previous article. So, please read our previous article before proceeding to this article. Here, in this article, I will discuss how to create a Form in ASP.NET Core Application using the Form Tag Helpers.

**What are the Tag Helpers used to create a Form in ASP.NET Core?**

In order to create a Form in ASP.NET Core Application, we need to use the following common Tag Helpers.

1. **Form Tag Helper**
2. **Input Tag Helper**
3. **Label Tag Helper**
4. **Textarea Tag Helper**
5. **Radio Button Tag Helper**
6. **Select Tag Helper**

At the end of this article, we will create a Form as shown below using the above Tag Helpers. The following form is used to create a Student.



**Form Tag Helper in ASP.NET Core:**

In order to create a Form in ASP.NET Core MVC View, we need to use the <form> tag helper. The syntax to use the Form Tag Helper is shown below.



As you can see in the above image, within the Form Tag Helper, we are using the asp-controller and asp-action tag helpers. These two tag helpers specify the controller and the action method to which the form data is going to be submitted. The method type specifies whether it is a Get request or Post Request. We want to issue a Post request when the form is submitted, so we set the method type as Post.

**Note:** If you didn’t specify the controller and action name using the asp-controller and asp-action tag helpers, then by default, when the form is submitted, it will be invoked the same action method of the controller which rendered the form.

**Input Tag Helper in ASP.NET Core:**

The Input Tag Helper in ASP.NET Core binds an HTML <input> element to a model expression in the razor view. Here, we want a form to create a new Student. So, the model for our view is Student class and we can specify the model using the following directive.

**@model Student**

In order to capture the student name, we want to display a text box in our form. We also want that text box to bind with the Name property of the Student model class. This can be done very easily by using the asp-for Tag helper as shown below.

**<input asp-for=”Name”>**

As you can see here we set the value for the asp-for tag helper to the Name property of the Student model class. Here, you will also get the IntelliSense while setting the value property. Later if you change the property name, let say from Name to StudnetName on the Student class, and if you do not change the value assigned to the tag helper, then you will get a compiler error.

The above input tag helper generates an input element with id and name attributes. And both the id and name are set to a value as Name as shown below.

**<input type=”text” id=”Name” name=”Name” value=””>**

**Label Tag Helper in ASP.NET Core:**

The Label Tag Helper in ASP.NET Core generates a label with for attribute. The “for” attribute is used to link the label with its corresponding input element. For example,

**<label asp-for=”Name”></label>**  
**<input asp-for=”Name”>**

The above code generates the following HTML.

**<label for=”Name”>Name</label>**  
**<input type=”text” id=”Name” name=”Name” value=””>**

Here, the label is linked with the input element. This is because both the label for attribute and the input element id attribute have the same value (i.e. Name). That means when we click on the label, then the corresponding input element will receive the focus.

**TextArea Tag Helper in ASP.NET Core:**

The Textarea tag helper in ASP.NET Core is very much similar to the input tag helper but specifically targets the Textarea element instead of the input element. The textarea tag helper is used by adding the asp-for tag helper attribute to a text area element. For example, let say out Student having a property to store the address, then for address property, we can use textarea tag helper as shown below.

**<textarea asp-for=”Address”></textarea>**

The text area tag helper was able to generate name and id properties based on the name of the property specified in asp-for. If you want to display the textarea with a specified number of rows and cols, then you need to use the rows and cols attribute as shown below.

**<textarea asp-for=”Address” rows=”4″ cols=”30″></textarea>**

**Select Tag Helper in ASP.NET Core:**

The Select Tag helper in ASP.Net Core generates a select tag with its associated option elements. In our example, we want a select element to display the list of Branches. We also want a label which should be linked with the select element. The following code does the same.

**<label** for="Branch"**>**Branch**</label>**

**<select** id="Branch" name="Branch"**>**

**<option** value="0"**>**None**</option>**

**<option** value="1"**>**CSE**</option>**

**<option** value="2"**>**ETC**</option>**

**<option** value="3"**>**Mech**</option>**

**</select>**

The options for the branch select element can be hard-coded like in the above example, or they can also come from enum or even though from a database table. In our example, we are going to use an enum for the select options.

**Radio Button Tag Helper in ASP.NET Core:**

The radio button control is designed to support the selection of only one of a mutually exclusive set of predefined options. In order to generate radio button in ASP.NET Core application, we need to use the radio button tag helper.

**Let us create a Form using the above Form Tag Helpers:**

First, add the following **Create** action method within the Home Controller.

**[**HttpGet**]**

**public** ViewResult Create**()**

**{**

Student student = new Student

**{**

AllGenders = Enum.GetValues**(**typeof**(**Gender**))**.Cast**<**Gender**>()**.ToList**()**

**}**;

**return** View**(**student**)**;

**}**

**Creating the Create View:**

Now add a view with the name **Create.cshtml** within the Home folder of your application. Once you add the **Create.cshtml** view, then copy and paste the following code in it.

@model FirstCoreMVCApplication.Models.Student

@{

ViewBag.Title = "Create";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**<form** asp-controller="Home" asp-action="Create" method="post" class="mt-3"**>**

**<div** class="form-group row"**>**

**<label** asp-for="Name" class="col-sm-2 col-form-label"**></label>**

**<div** class="col-sm-10"**>**

**<input** asp-for="Name" class="form-control" placeholder="Name"**>**

**</div>**

**</div>**

**<div** class="form-group row"**>**

**<label** asp-for="Email" class="col-sm-2 col-form-label"**></label>**

**<div** class="col-sm-10"**>**

**<input** asp-for="Email" class="form-control" placeholder="Email"**>**

**</div>**

**</div>**

**<div** class="form-group row"**>**

**<label** asp-for="Branch" class="col-sm-2 col-form-label"**></label>**

**<div** class="col-sm-10"**>**

**<select** asp-for="Branch" class="custom-select mr-sm-2"

asp-items="Html.GetEnumSelectList<Branch**>**()">**</select>**

**</div>**

**</div>**

**<div** class="form-group row"**>**

**<label** asp-for="Gender" class="col-sm-2 col-form-label"**></label>**

**<div** class="col-sm-10"**>**

@foreach (var gender in Model.AllGenders)

{

**<label** class="radio-inline"**>**

**<input** type="radio" asp-for="Gender" value="@gender" id="Gender@(gender)" **/>**@gender**<br** **/>**

**</label>**

}

**</div>**

**</div>**

**<div** class="form-group row"**>**

**<label** asp-for="Address" class="col-sm-2 col-form-label"**></label>**

**<div** class="col-sm-10"**>**

**<textarea** asp-for="Address" class="form-control" placeholder="Address"**></textarea>**

**</div>**

**</div>**

**<div** class="form-group row"**>**

**<div** class="col-sm-10"**>**

**<button** type="submit" class="btn btn-primary"**>**Create**</button>**

**</div>**

**</div>**

**</form>**

Now, run the application and click on the Create Menu and you should see the required view as expected.

**Model Binding in ASP.NET Core**

**Model Binding in ASP.NET Core Application**

In this article, I am going to discuss **Model Binding in ASP.NET Core**Application. Please read our previous article where we discussed [**how to create a Form in ASP.NET Core Application**](https://dotnettutorials.net/lesson/form-tag-helpers-asp-net-core/) using Form Tag Helpers. Please read our previous article as we are going to work with the same application that we worked in our previous article.

**What is Model Binding in ASP.NET Core?**

The Model Binding is a mechanism in ASP.NET Core Application which extracts the data from an HTTP request and provides them to the controller action method parameters.

The action method parameters may be simple types like integers, strings, etc. or complex types such as Student, Order, Product, etc.

**How does the Model Binding works in ASP.NET Core?**

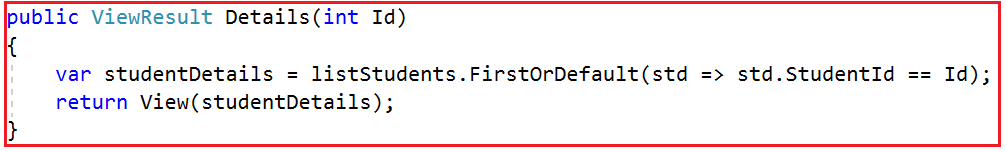
As we already discussed, it is our controller action method which is going to handle the incoming HTTP Request in ASP.NET Core MVC Application.

**Example using Route Data:**

Let us understand this with an example. When we want to view the details of a student whose id is 101, then we generally issue a GET request to the following URL.

**http://localhost:52191/home/details/101**

Our application default route template **({controller=Home}/{action=Index}/{Id?})** routes the above GET request to the **Details(int Id)**action method of the **HomeController**. The following image shows the Details action method of the Home Controller.

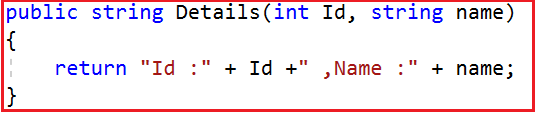


So, the value 101 in the request URL is mapped to the Id parameter of the Details(int Id) action method of Home Controller. The MVC Framework will automatically bind the data in the request to the action method parameters by name.

If you notice, the parameter name in the default route template is “Id” and the parameter name of the Details(int Id) action method is also “Id”. So the value 101 in the URL (<http://localhost:52191/home/details/101>) is mapped to the Id parameter of the Details(int Id) action method.

**Example using Query String:**

Let us understand this with an example. First, modify the Details Action method as shown below. As you can see we made two changes here. First, we change the return type of the action method to string. Secondly, the Details method now taking two parameters.



Now issue a Get Request as shown below.

**http://localhost:52191/home/details/101?name=dotnet**

The above GET request will handle by the Details action method and it will map the value 101 to the Id parameter and the value dotnet will be mapped to the name parameter of the Details action method.

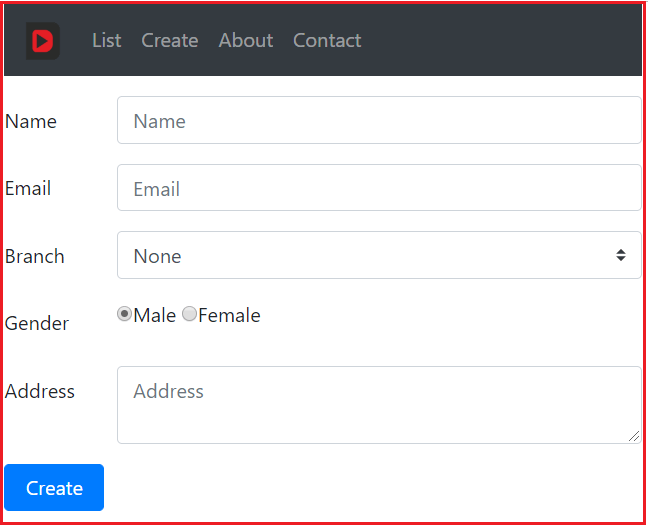
**HTTP Request Data Sources:**

ASP.NET Core MVC uses three primary data sources to map the HTTP requests data to the action method parameter in the following order:

1. **Form values:** Values in the FORM in HTTP POST requests.
2. **Route values:** Values provided by the Routing system.
3. **Query string:** Values found in the URL’s query string (e.g. after ? character).

**Model Binding in ASP.NET Core with Complex Type:**

The Model Binding in ASP.NET Core Application also works with complex types like Customer, Student, Order, Product, etc. Let us understand this with an example. In the previous article, we created the following Create Student form.



Add the following Create method to the Home Controller. When the above form is posted, this is method which is going to handle the request. Please decorate the method with HttpPost attribute.

**[**HttpPost**]**

**public** ActionResult Create**(**Student student**)**

**{**

student.StudentId = listStudents.Max**(**x =**>** x.StudentId**)** + 1;

listStudents.Add**(**student**)**;

**return** View**(**"Details", student**)**;

**}**

**How does it work?**

When the form is submitted, the values in the form are mapped to the Student object parameter to the Post Create action method. The Model binder in asp.net core application binds the posted form values to the properties of the Student object that is passed as a parameter to the Create() action method.

The value in the input element that has the name attribute set to “Name” is mapped to the Name property of the Studnet object. Similarly, the value in the Branch input element will be mapped to the Branch property of the Student object. This is going to be same for the rest of the properties like Email and Gender.

Note: At the moment if you navigate to list view, then you will not find the newly created student data. In a later article, we will discuss how to solve this issue when we are working with the database.

At the moment we don’t have any validation on the Create Student Form. So, if we submit the form without filling any of the form fields, then we will end up creating a new student with empty data.