Building Web Application using ASP.NET Core MVC

Objectives

- Overview ASP.NET Core
- List the advantages of ASP.NET Core
- Explain about WebServer
- Overview ASP.NET MVC Architecture
- Explain role of the Model, View and Controller
- Explain about ViewBag, ViewData, TempData and Session
- Explain about HTML Helper
- Explain about Model Binding and Model Validation
- Demo create ASP.NET Core MVC application with Entity Framework

ASP.NET Core Overview

Introduction to ASP.NET

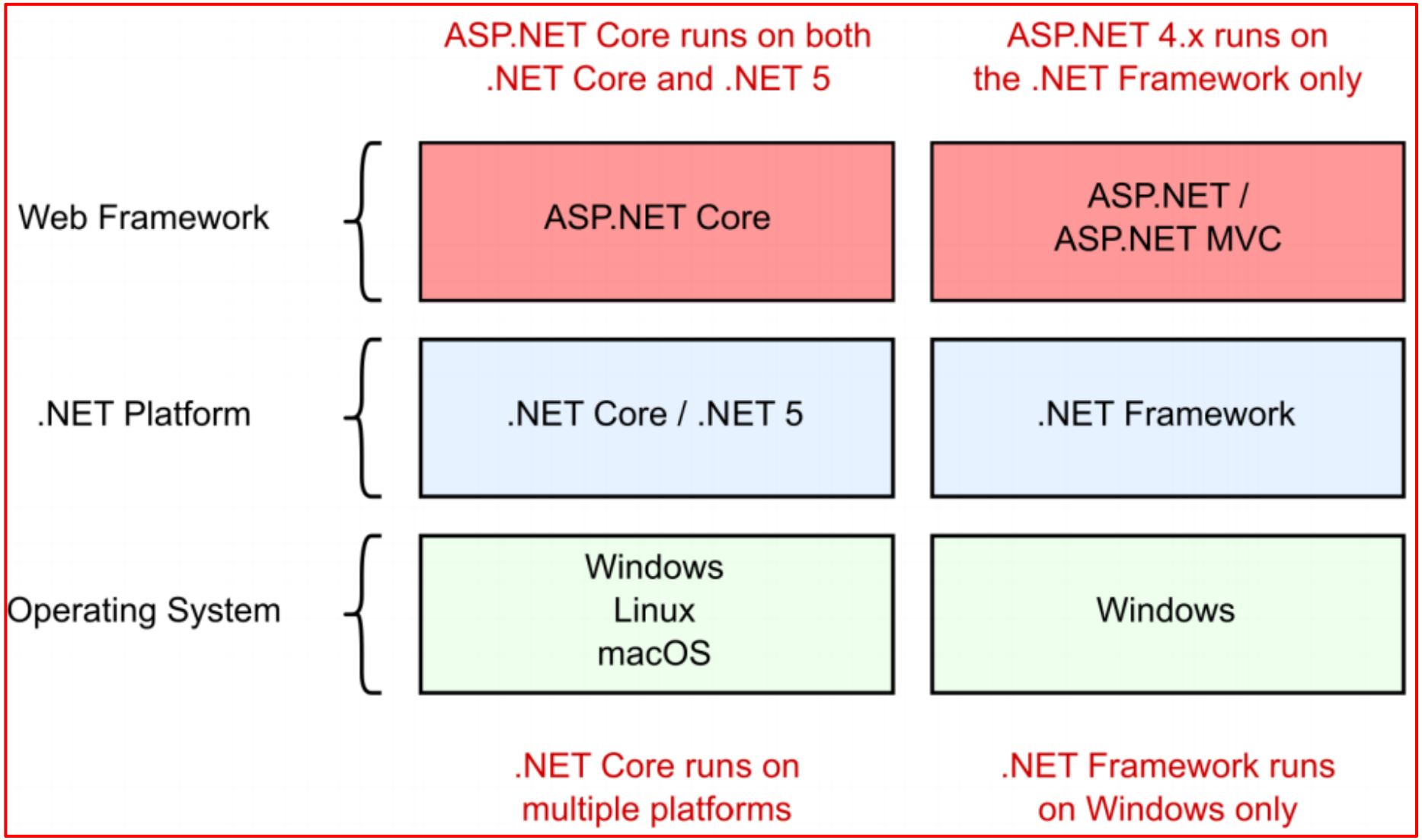
- ASP.NET is a web application framework designed and developed by Microsoft. ASP.NET is open source and a subset of the .NET Framework and successor of the classic ASP(Active Server Pages)
- With version 1.0 of the .NET Framework, it was first released in January 2002. Before .NET and ASP.NET Web Form there was Classic ASP
- ASP.NET is built on the CLR (Common Language Runtime) which allows the programmers to execute its code using any .NET language (C#, VB etc.)
- It is specially designed to work with HTTP and for web developers to create dynamic web pages, web applications, web sites, and web services as it provides a good integration of HTML, CSS, and JavaScript

The Limitations of ASP.NET

- ASP.NET Web Forms suffered from many issues, especially when building larger applications. In particular, a lack of testability, a complex stateful model, and limited influence over the generated HTML (making client-side development difficult) led developers to evaluate other options
- Auto generated HTML does not provide full control to the developers
- HTML ID management is compromised and it makes difficult to use client side languages like Jquery

What is the ASP.NET Core?

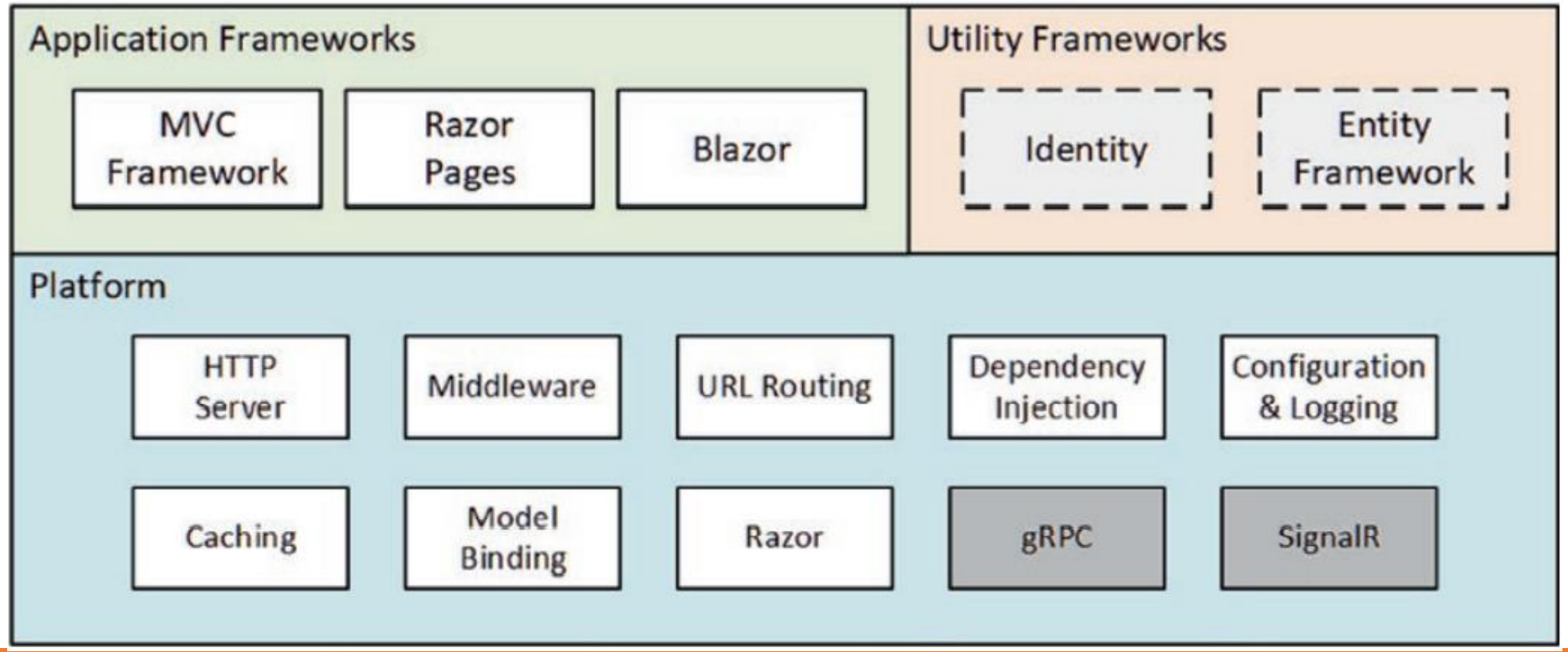
- ASP.NET Core is the new version of the ASP.NET web framework mainly targeted to run on .NET Core platform
- ASP.NET Core is a free, open-source, and cross-platform framework for building cloud-based applications, such as web apps, IoT apps, and mobile backends. It is designed to run on the cloud as well as on-premises
- Same as .NET Core, it was architected modular with minimum overhead, and then other more advanced features can be added as NuGet packages as per application requirement. This results in high performance, require less memory, less deployment size, and easy to maintain
- ASP.NET Core is an open source framework supported by Microsoft and the community, so we can also contribute or download the source code from the ASP.NET Core Repository on Github



The relationship between ASP.NET Core, ASP.NET, .NET Core, and .NET Framework
ASP.NET Core runs on .NET Core, so it can run cross-platform

The Structure of ASP.NET Core

- ASP.NET Core consists of a platform for processing HTTP requests, a series of principal frameworks for creating applications, and secondary utility frameworks that provide supporting features, as illustrated by the following figure:



Features of ASP.NET Core

- Streamlined Web development
- A system that is set to work on cloud
- Good community base
- An integrated platform for creating a variety of Web applications and APIs
- Assimilation of latest frameworks
- Support for a flexible and lightweight HTTP request channel
- Support for hosting itself in a targeted process or on different platforms
- Simultaneous versioning of applications
- ASP.NET Core also allows us to create applications that follow the MVC architectural style, with a ready-made template that is available for use

Features of ASP.NET Core

- Support build HTTP-based web services as well as RESTful services. A new addition to the capability in implementing microservices by the **gRPC** template
- ASP.NET Core fully supports Razor, which contains an efficient language for creating our views and Tag Helpers, which allow logic to be written from the server side to generate HTML that can be used in Razor views
- Blazor WebAssembly: Blazor apps that were rendered server-side. They can also be rendered client-side, enabling offline and standalone apps
- Multi-Platform Web Apps: Blazor apps was originally conceived as a vehicle for web apps and works great in a browser. The goal is that this will work equally great for a mobile device, or a native desktop application

ASP.NET Core Advantages

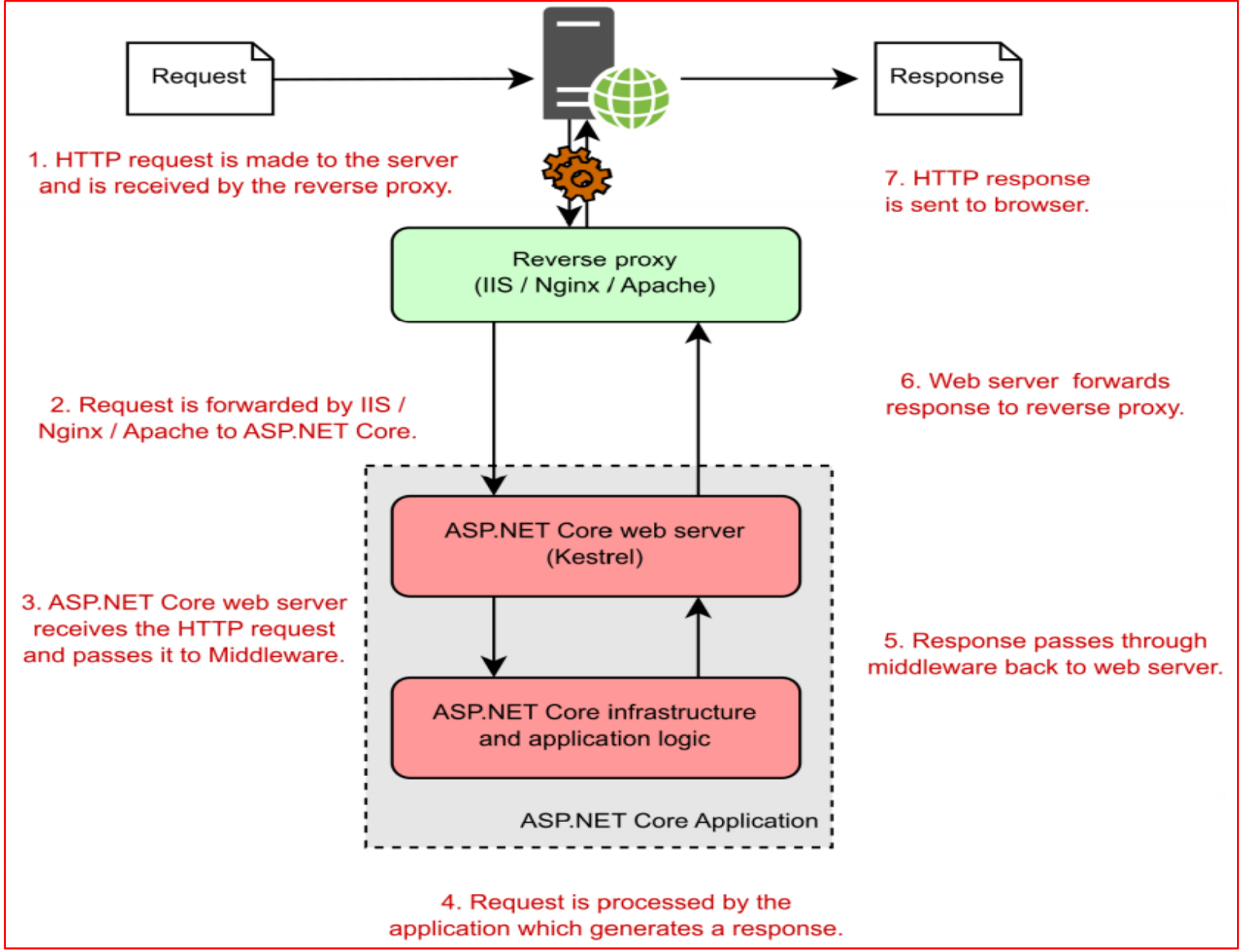
- Fast: ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages that we need for our application. This reduces the request pipeline and improves performance and scalability
- IoC Container: It includes the built-in IoC container for automatic dependency injection which makes it maintainable and testable
- Integration with Modern UI Frameworks: It allows us to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap, etc. using Bower (a package manager for the web)

ASP.NET Core Advantages

- Hosting: ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework
- Code Sharing: It allows you to build a class library that can be used with other .NET frameworks such as .NET Framework 4.x or Mono (a single code base can be shared across frameworks)
- Side-by-Side App Versioning: ASP.NET Core runs on .NET Core, which supports the simultaneous running of multiple versions of applications
- Smaller Deployment Footprint: ASP.NET Core application runs on .NET Core, which is smaller than the full .NET Framework. So, the application which uses only a part of .NET CoreFX will have a smaller deployment size. This reduces the deployment footprint

WebServer in ASP.NET Core

- Prior versions of ASP.NET applications could only be deployed to Windows servers using Internet Information Services (IIS). ASP.Net Core can be deployed to multiple operating systems in multiple ways, including outside of a web server
- The highlevel options are as follows:
  - On a Windows server (including Azure) using IIS
  - On a Windows server (including Azure app services) outside of IIS
  - On a Linux server using Apache or NGINX
  - On Windows or Linux in a Docker container

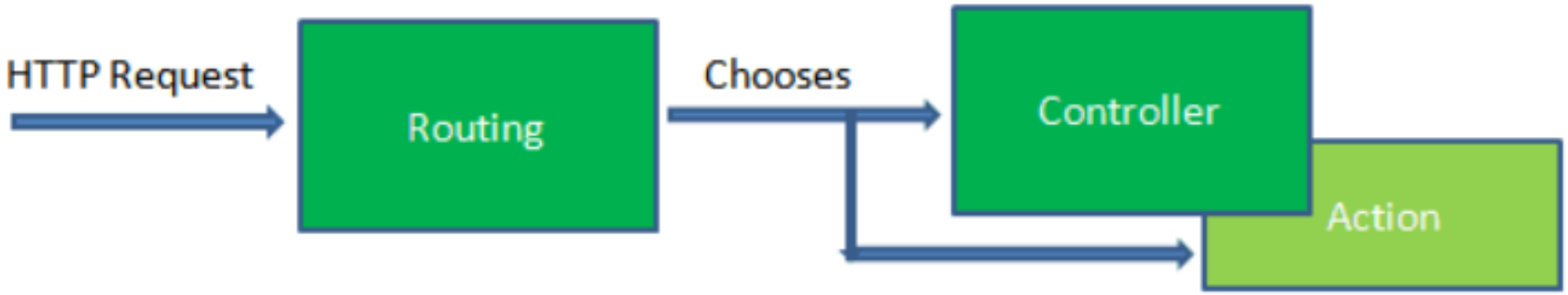


The hosting model for ASP.NET Core. Requests are received by the reverse proxy and are forwarded to the Kestrel web server. The same application can run behind various reverse proxies without modification

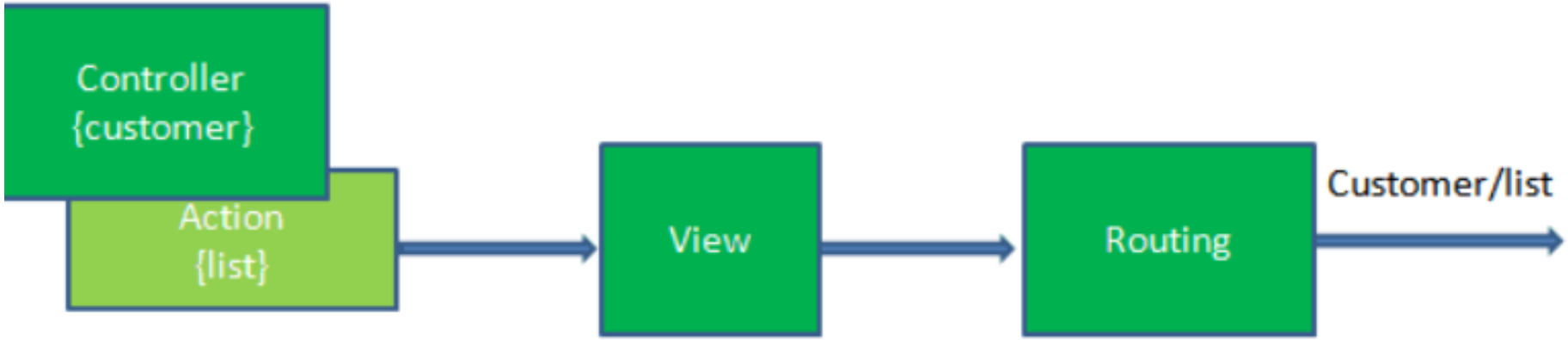
Routing in ASP.NET Core

- The Routing is the process by which ASP.NET Core inspects the incoming URLs and maps them to Controller Actions. It also used to generate the outgoing URLs
- This process is handled by the Routing Middleware. The Routing Middleware is available in Microsoft.AspNetCore.Routing Namespace
- The Routing has two main responsibilities:
  - It maps the incoming requests to the Controller Action
  - Generate an outgoing URLs that correspond to Controller actions

Routing in ASP.NET Core



Mapping the Incoming Requests



Constructing the URLs

Route and Route Handler

- The Route is similar to a roadmap. We use a roadmap to go to our destination. Similarly, the ASP.NET Core Apps uses the Route to go to the controller action
- The Each Route contains a Name, URL Pattern (Template), Defaults and Constraints. The URL Pattern is compared to the incoming URLs for a match. An example of URL Pattern is *{controller=Home}/{action=Index}/{id?}*
- The Route is defined in the Microsoft.AspNetCore.routing namespace
- The Route Handler is the Component that decides what to do with the route. When the routing Engine locates the Route for an incoming request, it invokes the associated RouteHandler and passes the Route for further processing
- The Route handler is the class which implements the IRouteHandler interface

How to Set up Routes

- There are two different ways by which we can set up routes:

1. Convention-based routing: use the **Configure** method of the **Startup** class
2. Attribute routing

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())...
    else...
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Name of Route

URL Pattern

Create ASP.NET Core MVC App by dotnet CLI

- Open **Command Prompt (or Terminal)** and run commands as follows:

```
D:\Demo\FU>dotnet new sln -o MySolution
The template "Solution File" was created successfully.
```

1. Create a new Solution named : MySolution

```
D:\Demo\FU>dotnet new mvc --no-https --output MySolution/MyWeb
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
```

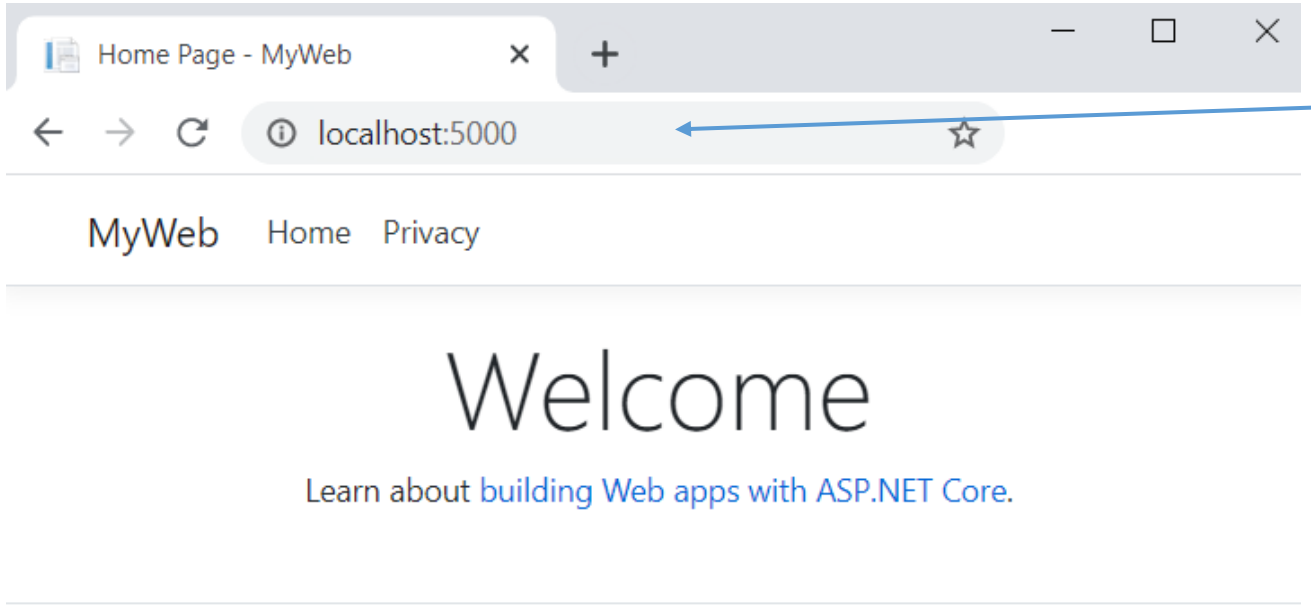
2. Create a new MVC project named MyWeb

```
D:\Demo\FU>dotnet sln MySolution add MySolution/MyWeb
Project `MyWeb\MyWeb.csproj` added to the solution.
```

3. Add project to Solution

```
D:\Demo\FU>dotnet run -p MySolution/MyWeb
Building...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\Demo\FU\MySolution\MyWeb
```

4. Run project : MyWeb

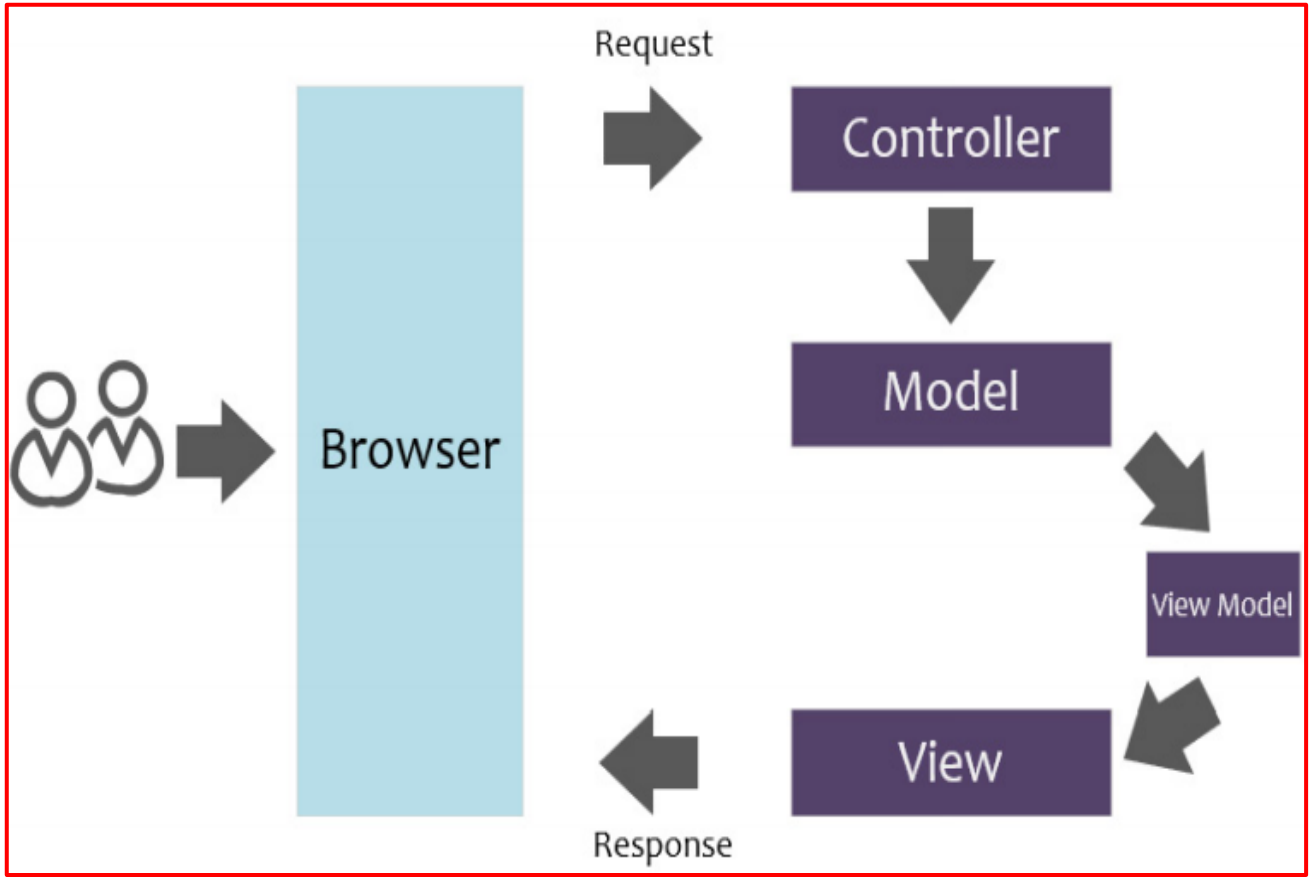


5. Access to web page

Working with ASP.NET Core MVC

Introducing the MVC Pattern

- The Model-View-Controller (MVC) pattern has been around since the 1970s, originally created as a pattern for use in Smalltalk
- The pattern has made a resurgence recently, with implementations in many different and varied languages, including Java (Spring Framework), Ruby (Ruby on Rails), and .NET (ASP.NET MVC)



The MVC request and response flow

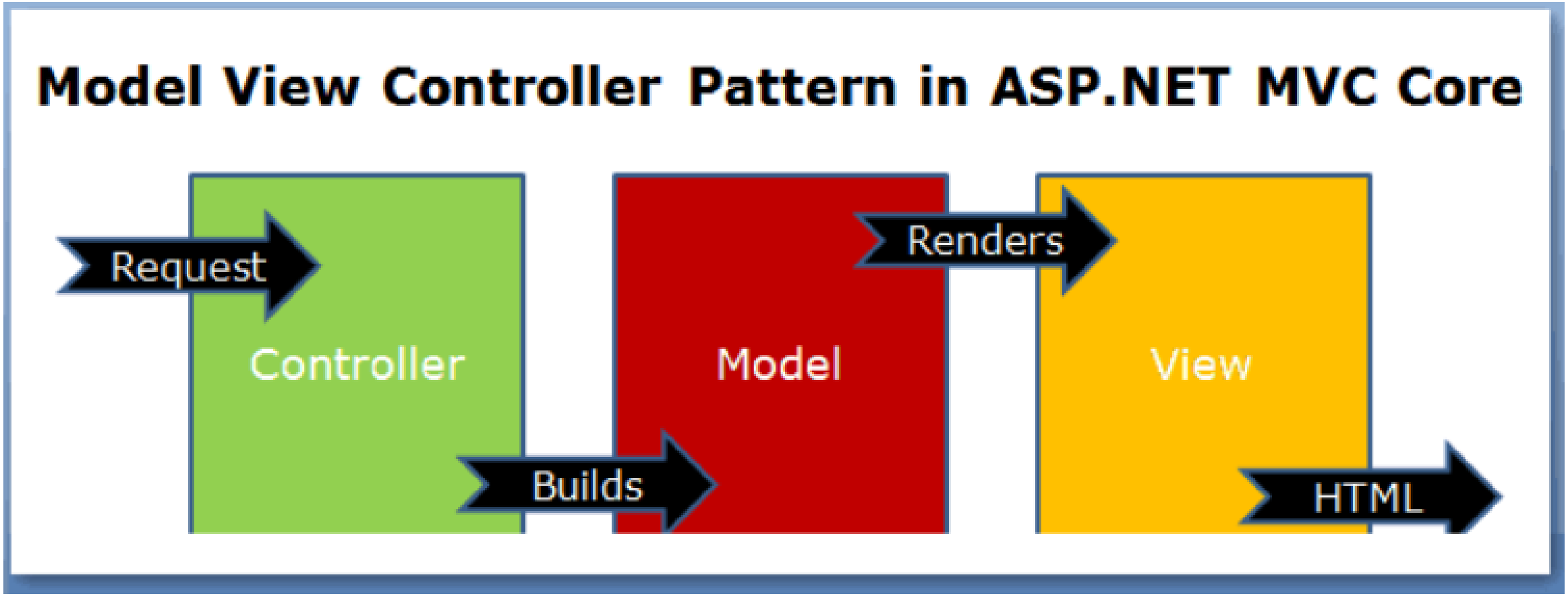
Introducing the MVC Pattern

- **Controllers** are responsible for handling any user interactions and requests, processing any logic that is required to fulfill the request, and ultimately, returning a response to the user. In other words, controllers orchestrate the flow of logic
- **Models** are components that actually implement domain-specific logic. Often, models contain entity objects, our business logic, and data access code that retrieves and stores data. We should consider separating our business logic and data access layer to value the separation of concerns and single responsibility

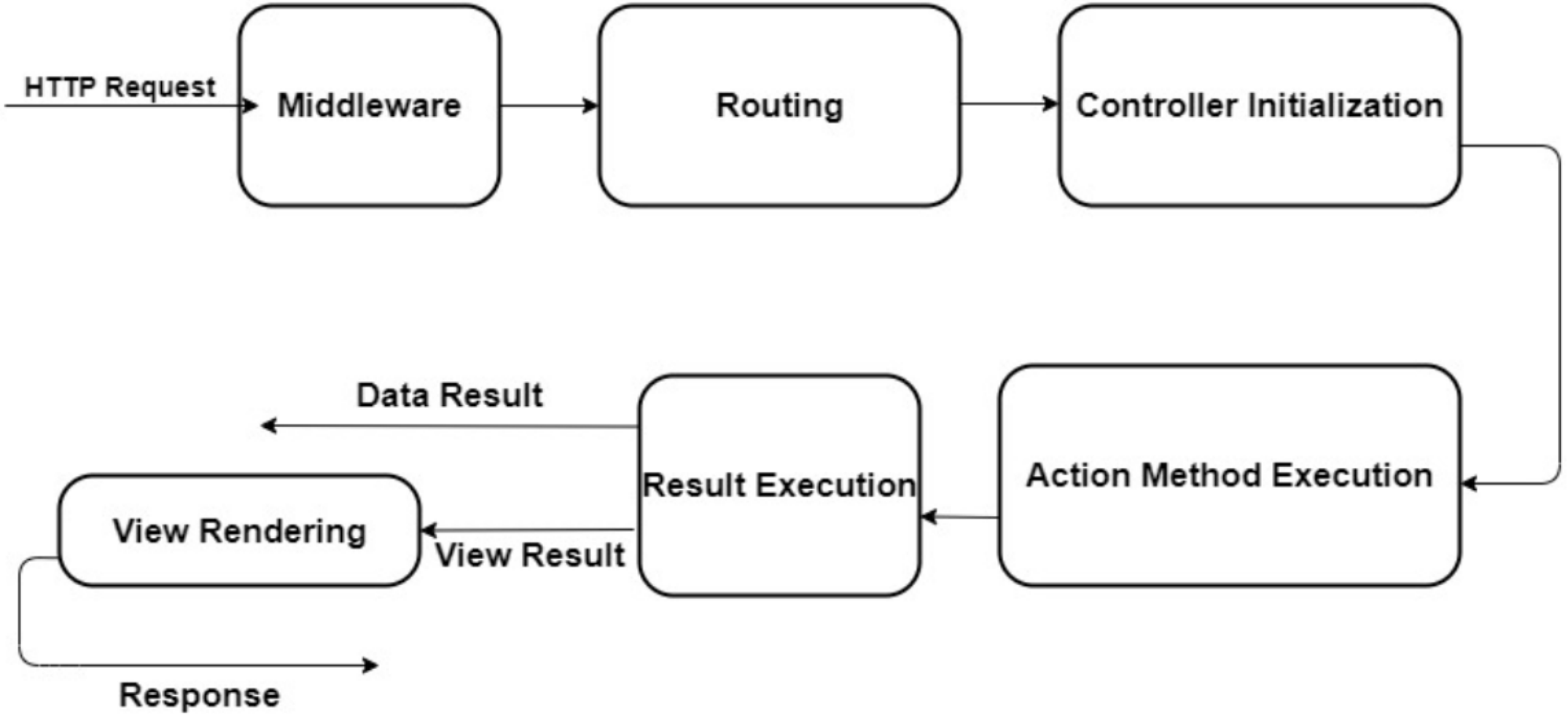
Introducing the MVC Pattern

- **Views** are components that make up our UI or page. Typically, views are just Razor files (.cshtml) that contain HTML, CSS, JavaScript, and C#-embedded code
- **ViewModel** is simply a class that houses some properties that are only needed for the view. ViewModel is optional because we can technically return a model to a view directly. It enables us to expose only the data that need instead of returning all data from entity object via models

How MVC Pattern works in ASP.NET Core

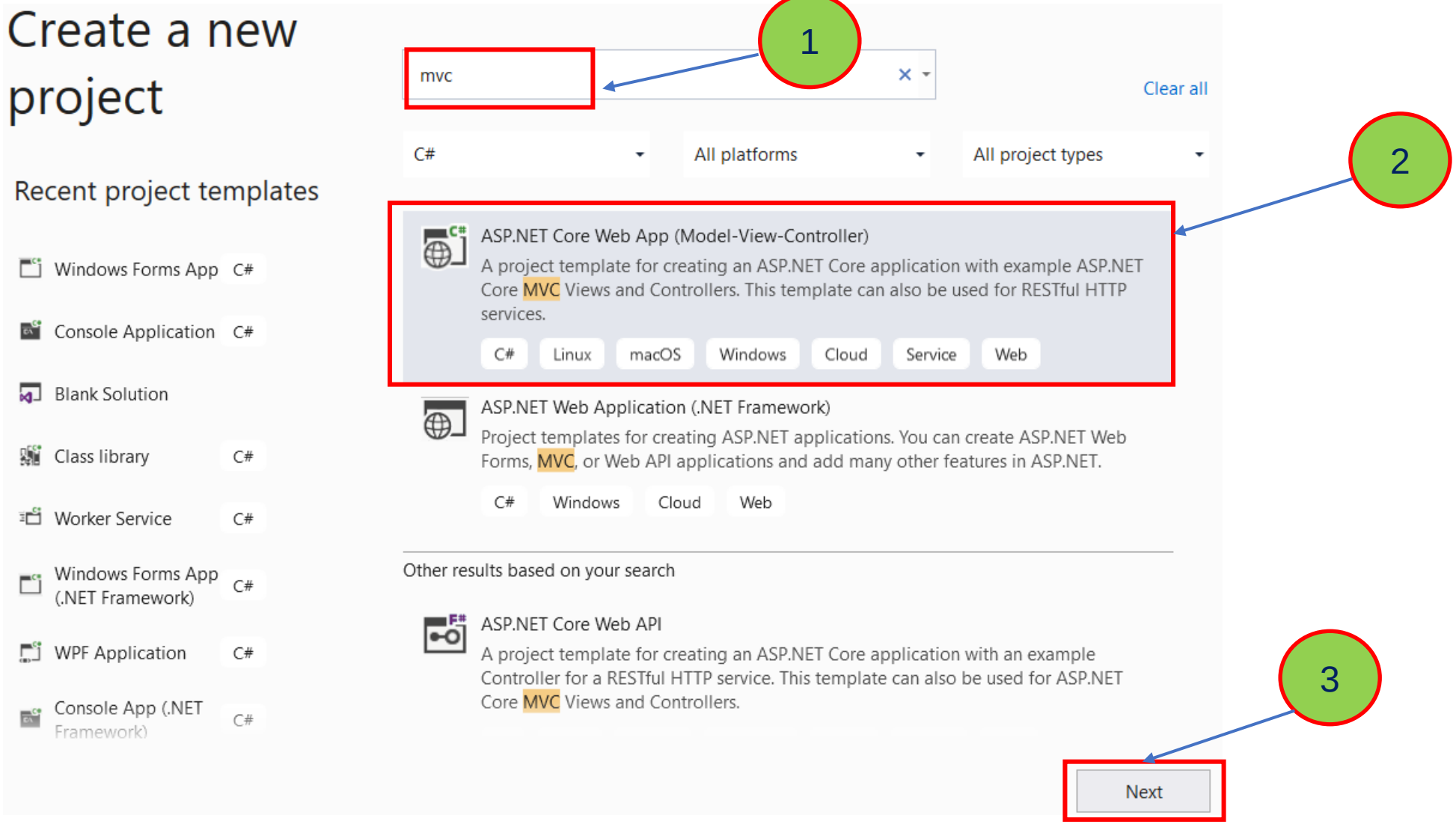


ASP.NET Core MVC Request Life Cycle

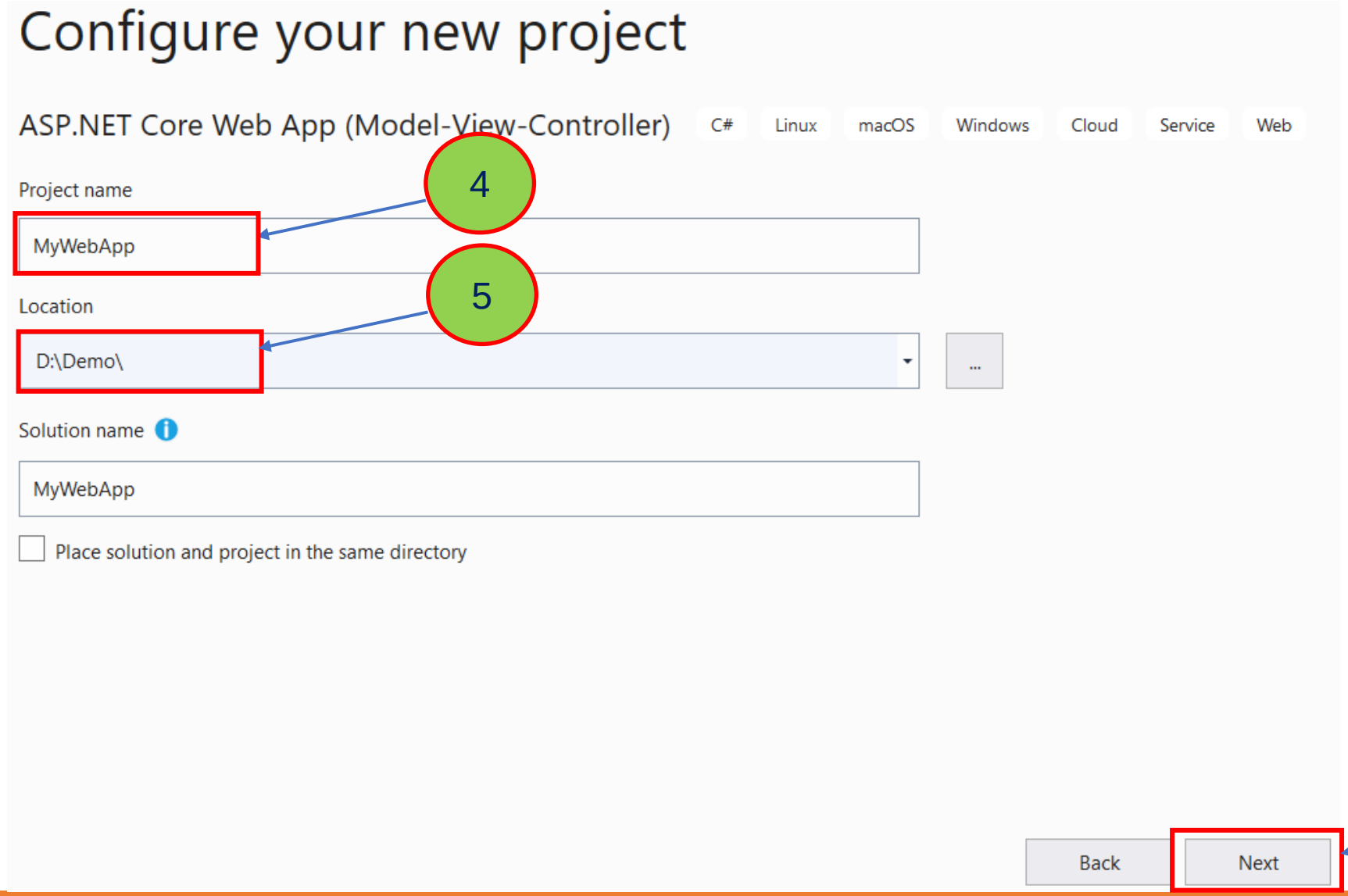


Demo 01: Create ASP.NET Core MVC Project using Visual Studio.NET

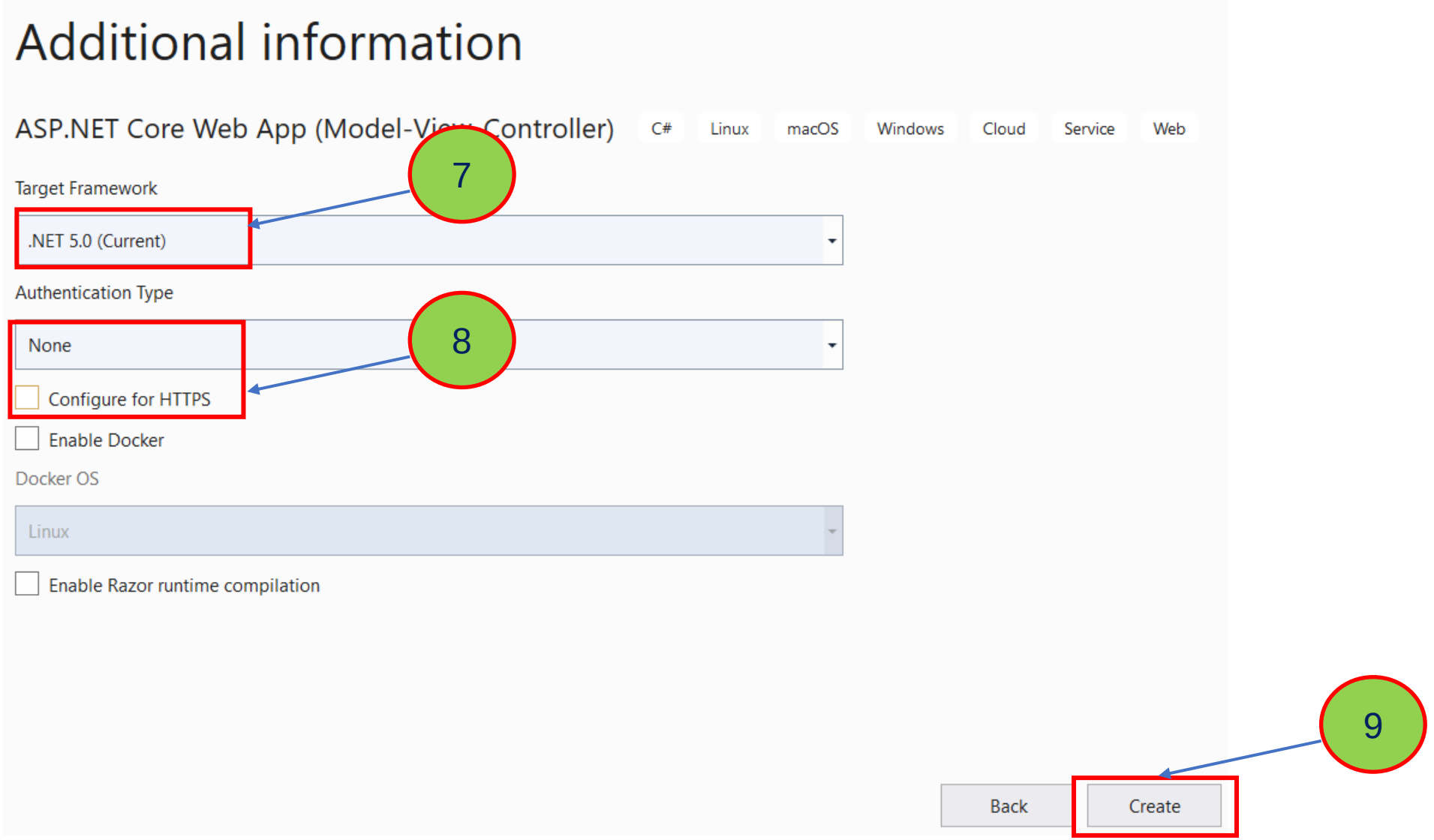
1. Open Visual Studio.NET , File | New | Project



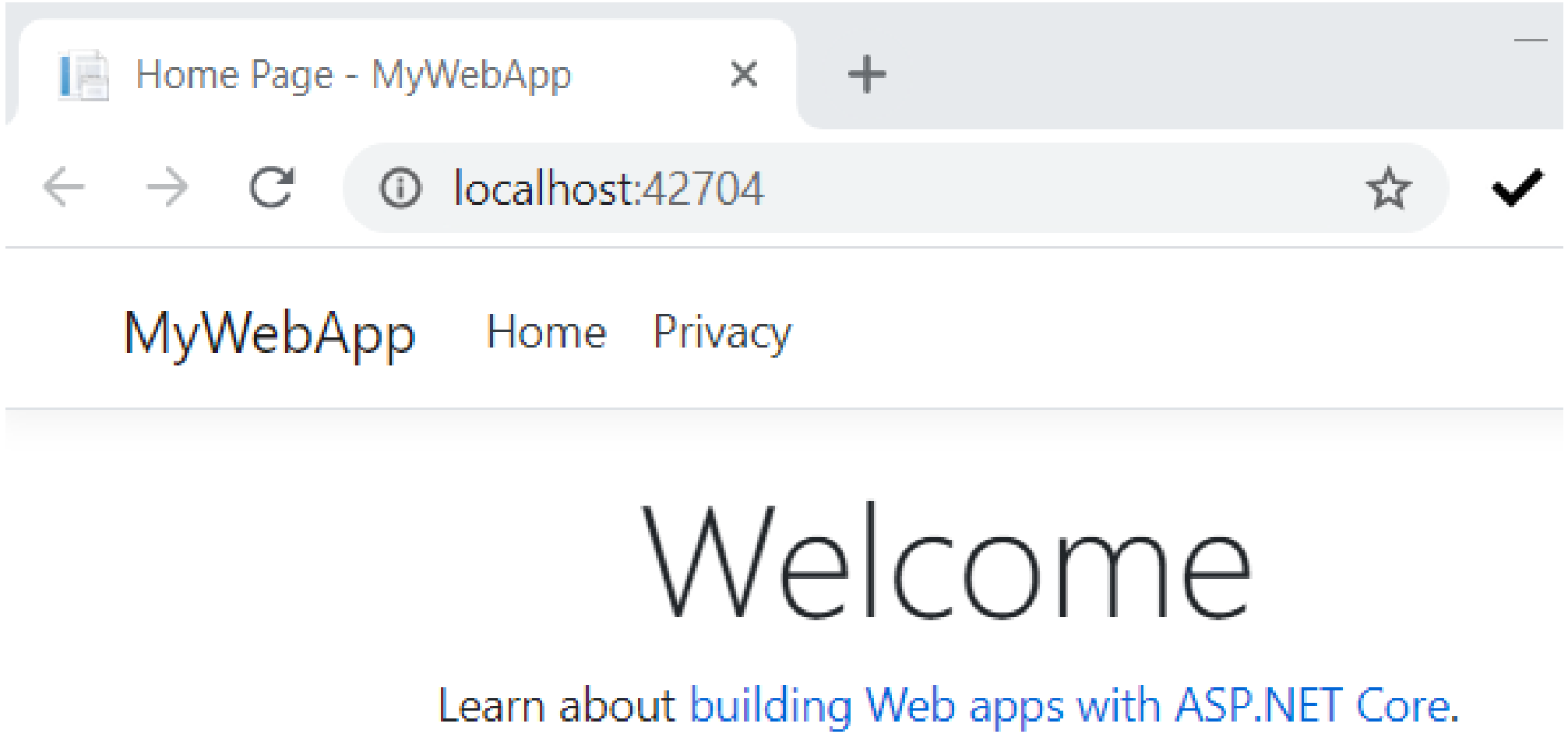
2. Fill out **Project name**: MyWebApp and **Location** then click **Next**



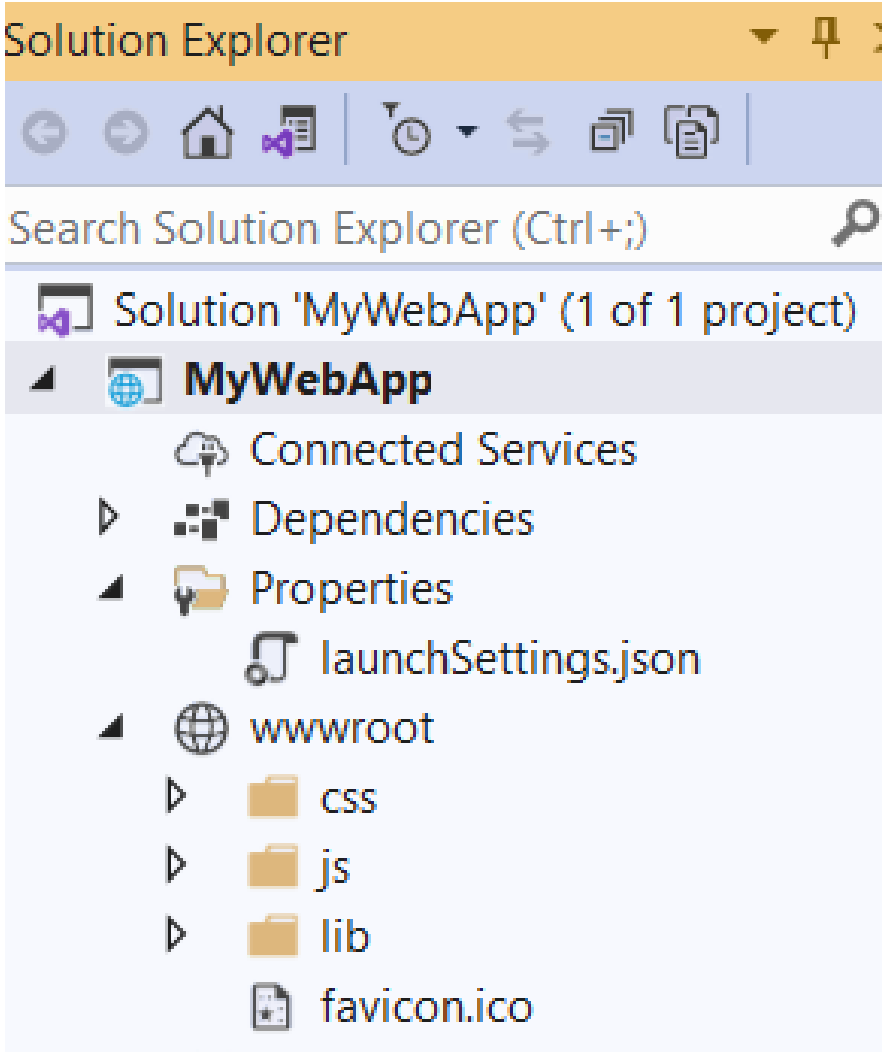
3. Config as follows then click **Create**

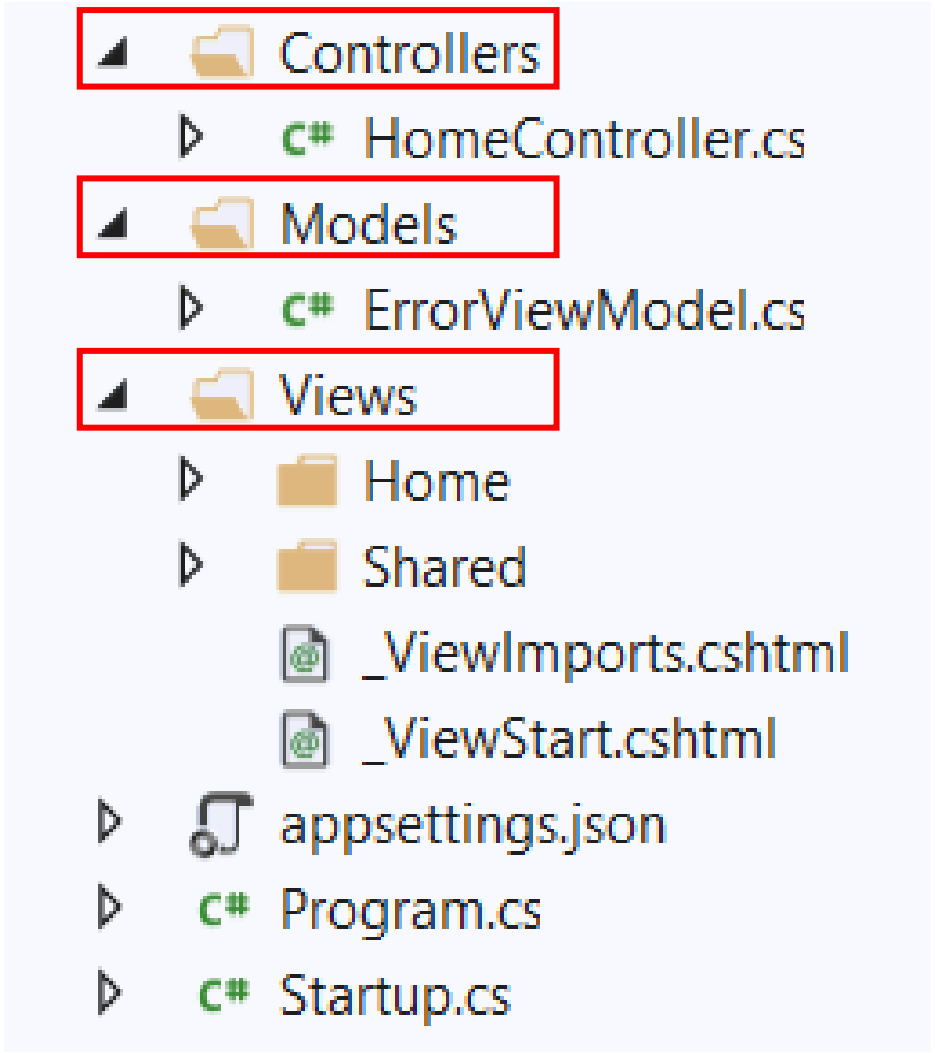


3. Press Ctrl+F5 to run project



Default MVC Project Structure





Default MVC Project Structure

- **Connected Services**: This allows us to connect to services such as Application Insights, Azure Storage, mobile, and other ASP.NET Core services that our application depends on, without we have to manually configure their connection and configurations
- **Dependencies**: This is where project dependencies are located, such as NuGet packages, external assemblies, the SDK, and framework dependencies needed for the application
- **Properties**: This folder contains the *launchSettings.json* file, where you can define application variables and profiles for running the app
- **wwwroot**: This folder contains all your static files, which will be served directly to the clients, including HTML, CSS, images, and JavaScript files

Default MVC Project Structure

- **appsettings.json**: This is where we configure application-specific settings. Keep in mind though that sensitive data should not be added to this file. We should consider storing secrets and sensitive information in a vault or secrets manager

- **Program.cs**: This file is the main entry point for the application. This is where we build the host for application. By default, the ASP.NET Core app builds a generic host that encapsulates all framework services needed to run the application

- **Startup.cs**: This file is the heart of any .NET application. This is where we configure the services and dependencies required for application

Default MVC Project Structure

- **The Controllers folder** is where the ASP.NET Core MVC and API implementations (and the routing engine) expect that the controllers for application are placed
- **The Views folder** is where the views for the application are stored. Each controller gets its own folder under the main Views folder named after the controller name (minus the Controller suffix). The action methods will render views in their controller's folder by default
  - For example, the Views/Home folder holds all the views for the HomeController controller class
- **The Shared Folder:** This folder is accessible to all controllers and their action methods. After searching the folder named for the controller, if the view can't be found, then the Shared folder is searched for the view

Controller Class

- Controller class inherited from the ControllerBase class
  - Some of the Helper Methods Provided by the Controller Class:

Helper Method	Description
ViewDataTempDataViewBag	Provide data to the view through the ViewDataDictionary, TempDataDictionary, and dynamic ViewBag transport
View	Returns a ViewResult (derived from ActionResult) as the HTTP response. Defaults to view of the same name as the action method, with the option of specifying a specific view. All options allow specifying a ViewModel that is strongly typed and sent to the View
PartialView	Returns a PartialViewResult to the response pipeline
ViewComponent	Returns a ViewComponentResult to the response pipeline
Json	Returns a JsonResult containing an object serialized as JSON as the response
OnActionExecuting	Executes before an action method executes
OnActionExecuted	Executes after an action method executes

Understanding Controllers

- The Controller in MVC architecture handles any incoming URL request and the name of the controller class must be suffixed with the name Controller
- Controllers in ASP.NET MVC inherited from the **Controller class**
- Controller class contains public methods called **Action** methods. Controller and its action method handles incoming browser requests, retrieves necessary model data and returns appropriate responses

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Understanding Action Method

- All the public methods of the Controller class are called Action methods. They are like any other normal methods with the following restrictions:
  - Action method must be public. It cannot be private or protected
  - Action method cannot be overloaded
  - Action method cannot be a static method

```
public class HomeController : Controller
{
    public IActionResult Index()=> View();

    public string Hello() => "Hello, ASP.NET Core MVC";
}
```

Understanding Action Method

- Miscellaneous Action Results

Action Method	Description
IActionResult	Defines a contract that represents the result of an action method.
ActionResult	A default implementation of IActionResult
ContentResult	Represents a text result
EmptyResult	Represents an ActionResult that when executed will do nothing
JsonResult	An action result which formats the given object as JSON
PartialViewResult	Represents an ActionResult that renders a partial view to the response
ViewResult	Represents an ActionResult that renders a view to the response
ViewComponentResult	An IActionResult which renders a view component to the response

Understanding Model

- The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it
- Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application
- Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model
- The Model is divided several categories based on how and where they are used. The Three main distinctions are: **Domain Model**, **View Model** and **Edit Model**

Understanding View

- The View is responsible for rendering the model to the Users
- The Controller gets the request and executes the appropriate business logic and gets the required data (model) then delegates the responsibility of rendering the model to the View
- Responsibilities of the Views
  - The rendering the data or model is the only responsibility of the View. The Views should not contain any logic and must not do any processing
  - The View can use any format to return to the user. The format could be HTML, JSON, XML or any other format for the user to consume as the response to the web request

Demo 02: Create Model-View-Controller (Reuse Demo-01)

1.Right-click on **Model** folder | Add | Class, named **HomeModel.cs** then write codes as follows:

```
namespace MyWebApp.Models
{
    public class HomeModel
    {
        public string Message = "Welcome to ASP.NET MVC Core";
    }
}
```

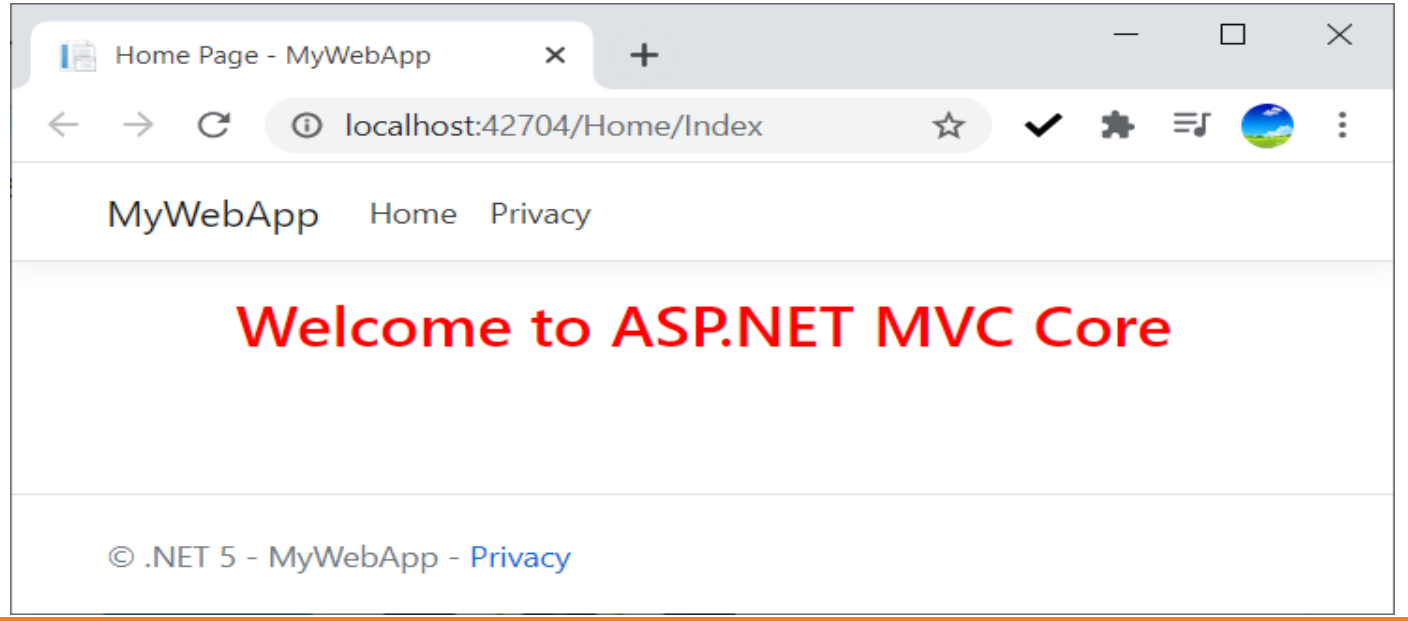
2.Open **HomeController.cs** then update code as follows:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        //Create HomeModel object
        HomeModel message = new HomeModel();
        //Invoke to the view with message object
        return View(message);
    }
}
```

3.Open **Index.cshtml** from **View | Home** folder then update as follows:

```
@model MyWebApp.Models.HomeModel;
@{
    ViewData["Title"] = "Home Page";
}
<div><div class="text-center">
    <p><h2 style="color:red">@Model.Message</h2></p>
</div></div>
```

4.Right-click on **Index.cshtml** view , select **View in Browser (**or press Ctrl+F5 to run project)



The Razor Syntax

- The Razor uses the @ symbol to transition from HTML markup to the C# code.The following are the two ways, by which you can achieve the transitions
  - **Using Razor code expressions** : started with @ and followed by C# code. The Code expression can be either Implicit or Explicit
  - **Using Razor code blocks**: started with @ symbol followed by curly braces and can use code blocks anywhere in the markup. A Razor code block can be used manipulate a model, declare variables, and set local properties on a view, etc
- These expressions are evaluated by the Razor View Engine and written to the response

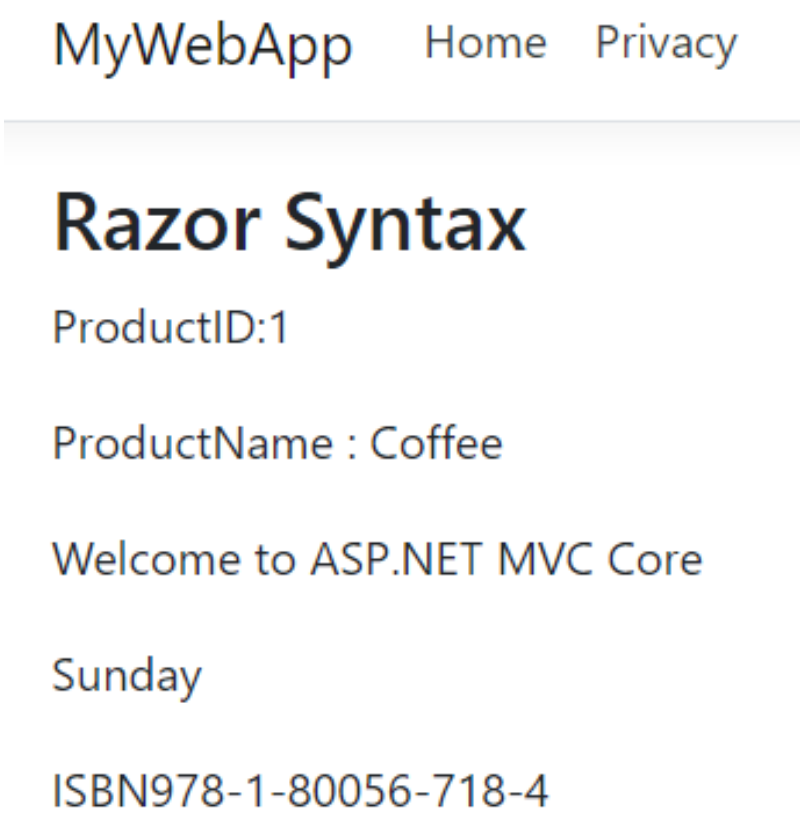
The Razor Syntax

Razor Code blocks

```
<h3>Razor Syntax</h3>
@{
    var greeting = "Welcome to ASP.NET MVC Core";
    var weekDay = DateTime.Now.DayOfWeek;
}
@{ var ISBNNo = "978-1-80056-718-4"; }
@{
    var pro = new { ProductID = 1, ProductName = "Coffee" };
    <p>ProductID:@pro.ProductID</p>
    <p>ProductName : @pro.ProductName</p>
    <p>@greeting</p>
    <p>@weekDay</p>
    //<p>ISBN@ISBNNo</p>    //Does not work
    <p>ISBN@(ISBNNo)</p>
}
```

Implicit Razor Expressions

Explicit Razor Expressions



Strongly Typed View

- The view which binds to a specific type of ViewModel is called as Strongly Typed View. By specifying the model, the Visual Studio provides the intellisense and compile time checking of type
- In the strongly typed View, we let the View know the type of ViewModel being passed to it, using the **@model** directive

```
@model StockWebApp.Models.Product

<form action="/home/edit" method="post">
    <input type="hidden" asp-for="ProductId" />
    <label for="@Model.ProductName">ProductName</label>
    <input type="text" asp-for="ProductName" />
    <label for="@Model.UnitPrice">UnitPrice</label>
    <input type="text" name="@Model.UnitPrice" />
    <label for="@Model.UnitsInStock">UnitsInStock</label>
    <input type="text" name="@Model.UnitsInStock" />
    <input type="submit" name="submit" />
</form>
```

Tag Helper

- The Tag helpers help us to write HTML elements in razor markup using easy to use syntax
- They look just like standard HTML code but it is processed by Razor engine on the server giving it all the advantageous of server-side rendering
- More Tag Helper:

https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-5.0

```
@model StockWebApp.Models.Product

<form asp-controller="Home" asp-action="Create">

    <label asp-for="ProductName"></label>
    <input asp-for="ProductName" />

    <label asp-for="UnitPrice"></label>
    <input asp-for="UnitPrice" />

    <label asp-for="UnitsInStock"></label>
    <input asp-for="UnitsInStock" />
    <input type="submit" name="submit" />

</form>
```

Model Binding

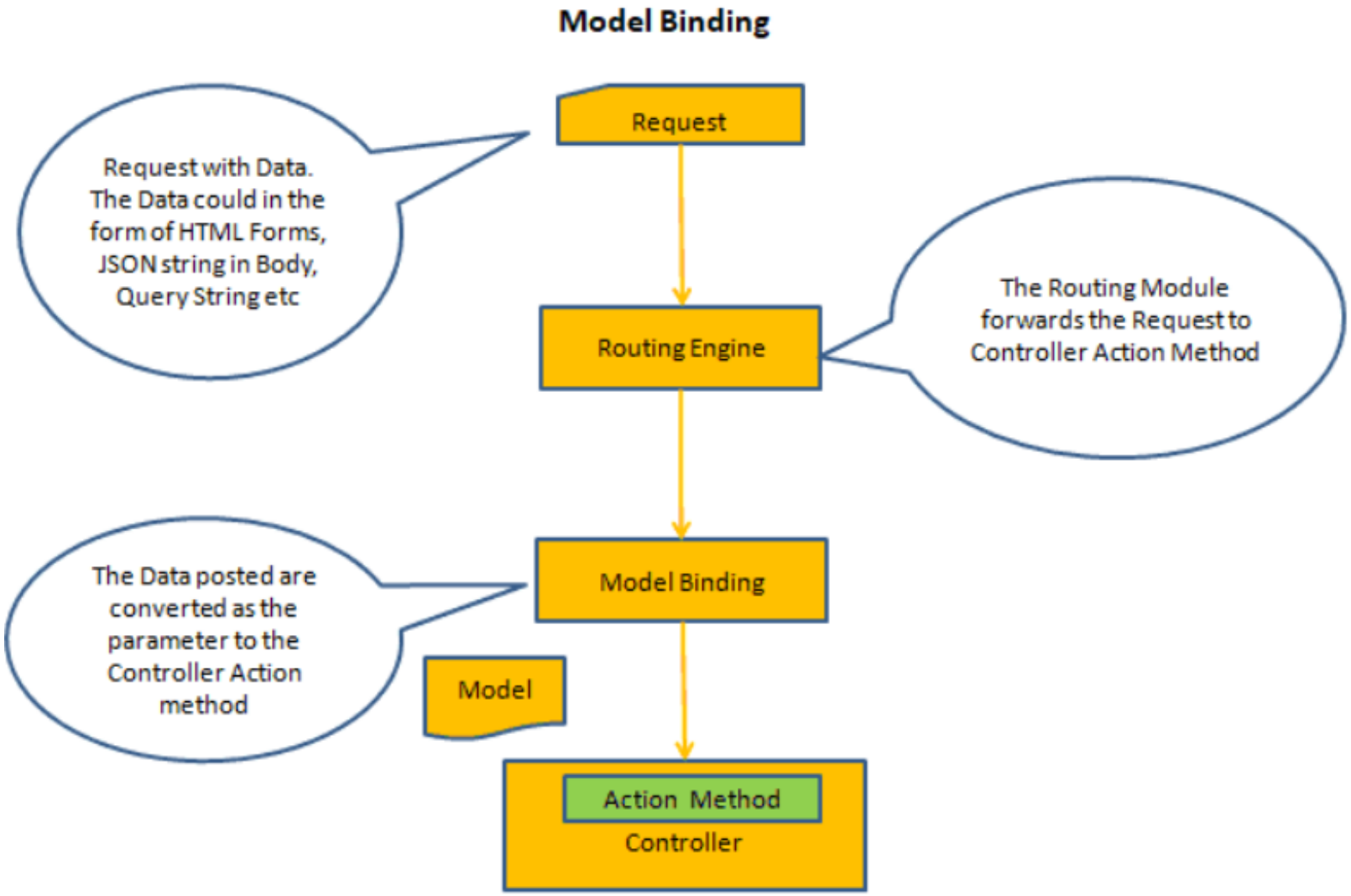
- The Model binding is the process of mapping the data posted over an HTTP request to the parameters of the action method in the Controller
- The HTTP Request can contain data in various formats. The data can contain in the HTML form fields. It could be part of the route values. It could be part of the query string or it may contain in the body of the request
- ASP.NET Core model binding mechanism allows us easily bind those values to the parameters in the action method. These parameters can be of the primitive type or complex type

```
public class ProductEditModel
{
  public int ID{ get; set; }
  public string Name { get; set; }
  public decimal Rate { get; set; }
  public int Rating { get; set; }
}
```

```
<form action="/home/Create" method="post">
    <label for="Name">Name</label>
    <input type="text" name="Name" />
    <label for="Rate">Rate</label>
    <input type="text" name="Rate" />
    <label for="Rating">Rating</label>
    <input type="text" name="Rating" />
    <input type="submit" name="submit" />
</form>
```

```
[HttpPost]
public IActionResult Create(ProductEditModel model){
    string message = "";
    if (ModelState.IsValid) {
        message = "product " + model.Name + " created successfully" ;
    }
    else{
        message = "Failed to create the product. Please try again";
    }
    return Content(message);
}
```

How Model Binding works



Model Validation

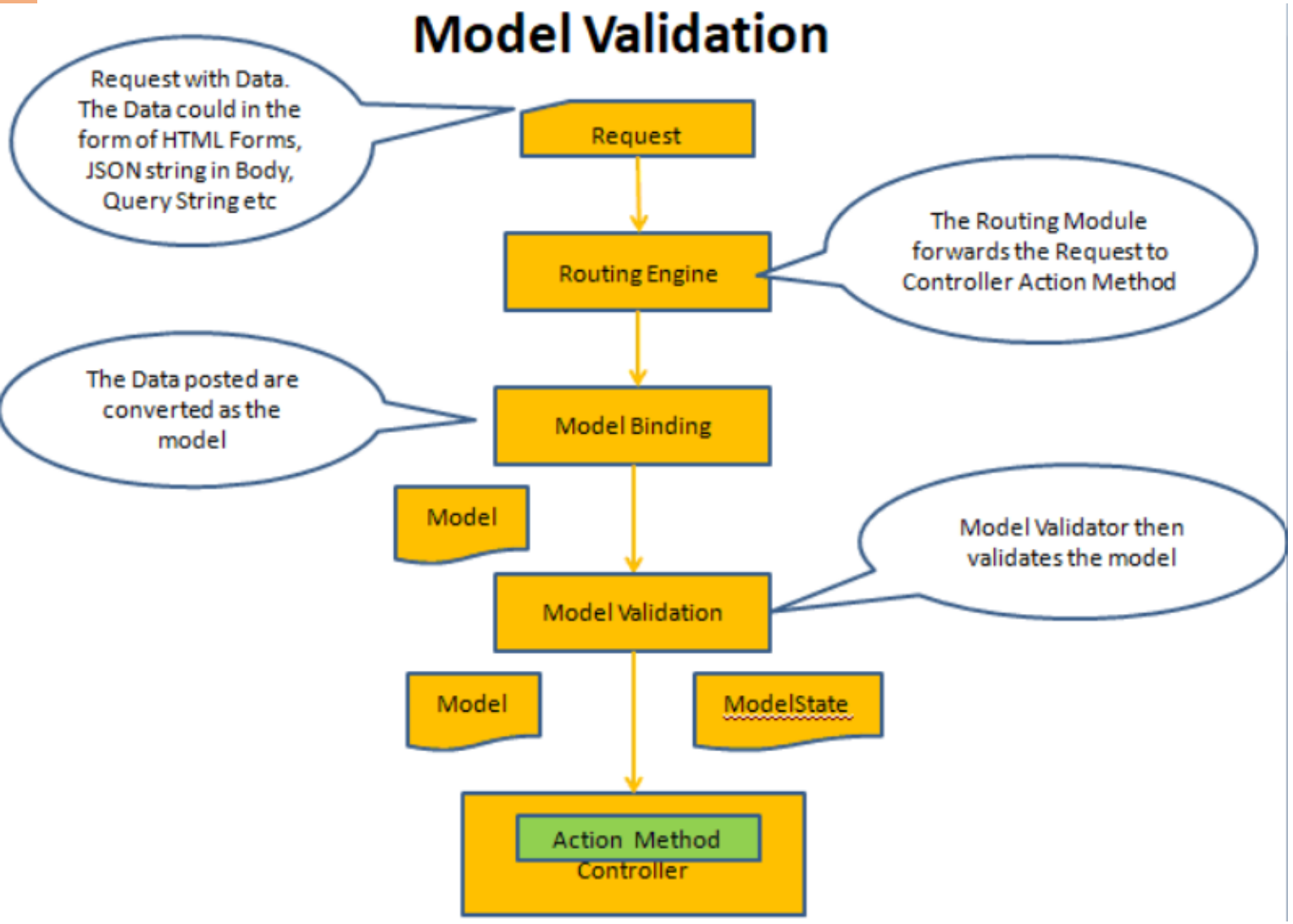
- Model state represents errors that come from two subsystems: model binding and model validation
- Errors that originate from model binding are generally data conversion errors. For example, an "x" is entered in an integer field. Model validation occurs after model binding and reports errors where data doesn't conform to business rules
- Both model binding and model validation occur before the execution of a controller action or a Razor Pages handler method

```
[Required(AllowEmptyStrings =false,ErrorMessage ="Please enter the name")]
[StringLength(maximumLength:25,MinimumLength =10,ErrorMessage ="Length must be between 10 to 25")]
public string Name { get; set; }
```

Model Validation

- [CreditCard]: Validates that the property has a credit card format. Requires jQuery Validation Additional Methods
- [Compare]: Validates that two properties in a model match
- [EmailAddress]: Validates that the property has an email format
- [Phone]: Validates that the property has a telephone number format
- [Range]: Validates that the property value falls within a specified range
- [RegularExpression]: Validates that the property value matches a specified regular expression
- [Required]: Validates that the field is not null. See [Required] attribute for details about this attribute's behavior
- [StringLength]: Validates that a string property value doesn't exceed a specified length limit
- [Url]: Validates that the property has a URL format
- [Remote]: Validates input on the client by calling an action method on the server

How Model Validation works




```
public class Movie {
    public int Id { get; set; }
    [Required]
    [StringLength(100)]
    public string Title { get; set; }
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }
    [Required]
    [StringLength(1000)]
    public string Description { get; set; }
    [Range(0, 999.99)]
    public decimal Price { get; set; }
}
```

Session and State Management

- HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. We can use several approaches to preserve user data between requests as follows:

Storage approach	Storage mechanism
Cookies	HTTP cookies. May include data stored using server-side app code
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code

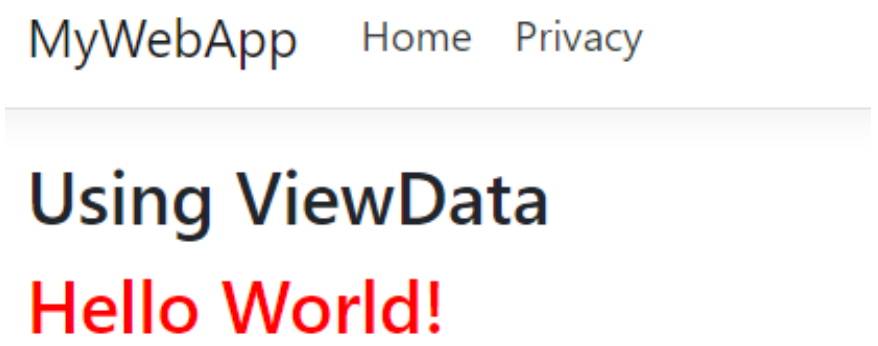
ViewData

- View Data is one of the most common and popular technique with the help of which we can pass the data from the controller to view
- Normally, view data is actually representing a dictionary which contains a key/value pair
- The below code demonstrates the controller method of StudentIndex which will return the view along with data

```
public IActionResult Index() {
    ViewData["Greeting"] = "Hello World";
    return View("Index");
}
```

Index.cshtml*

```
1  <h3>Using ViewData</h3>
2
3  <h3 style="color:red">@ViewData["Greeting"]!</h3>
4
```



ViewBag

- View Bag is quite similar to the ViewData. The main concept of the view bag is an instance of dynamic property

```
public class ProductModel {
    public int ProductID { get; set; }
    public string Name { get; set; }
    public string Brand { get; set; }
    public double Price { get; set; }
}
```

ProductModel

```
public IActionResult Index(){
    ViewBag.Message = "Welcome to ASP.NET Core";
    ViewBag.Product = new ProductModel{
        ProductID = 1,
        Name = "Samsung Galaxy Note",
        Brand = "Samsung",
        Price = 19000
    };
    return View();
}
```

ProductController

View: Index

```
<h2>Using ViewBag</h2>
<h3 style="color:red">@ViewBag.Message</h3>
<br>
@ViewBag.Product.ProductID <br>
@ViewBag.Product.Name <br>
@ViewBag.Product.Brand <br>
@ViewBag.Product.Price <br>
```

Using ViewBag

Welcome to ASP.NET Core

1
Samsung Galaxy Note
Samsung
19000

TempData

- Temp data is one of another data passing techniques from the controller method to view
- Temp data always return a data dictionary so that it can be derived from TempDataDictionary class
- It is worked as a temporary data storage. It will keep the data at the time of the HTTP request. Temp data is most useful to transfer data between different action methods in the different controllers. In the internal mechanism, temp data is basically used session variables
- It is mainly used to store data as a one-time message. We can keep the Tempdata source after the view is rendered by using the TempData.Keep() method

TempData

```
public IActionResult Index()
{
    List<string> lstStudent = new List<string>();
    lstStudent.Add("John");
    lstStudent.Add("Samrat");
    lstStudent.Add("David");
    TempData["StudentData"] = lstStudent;
    return View();
}
```

```
<h3>Using TempData</h3>
<ul>
    @foreach (var data in TempData["StudentData"] as List<string>)
    {
        <li>@data</li>
    }
</ul>
```

MyWebApp Home Privacy

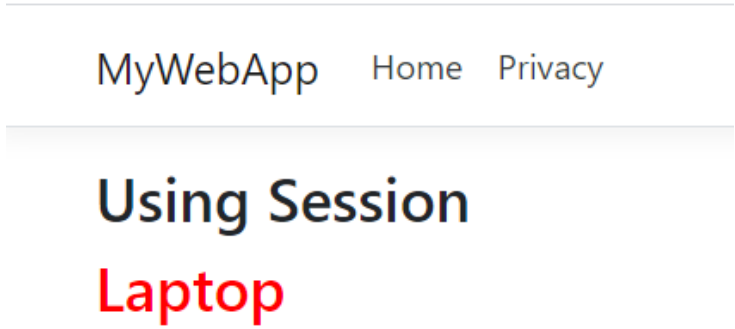
Using TempData

- John
- Samrat
- David

Session

- Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app
- Session state uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data

```
public IActionResult Index()
{
    HttpContext.Session.SetString("Product", "Laptop");
    return View();
}
[Route("Home/DemoSession")]
public IActionResult DemoSession()
{
    ViewBag.Data = HttpContext.Session.GetString("Product");
    return View();
}
```



Cookies

- Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum
- Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes
- Cookies are key-value pair collections where we can read, write and delete using key

```
//Set the cookie
CookieOptions option = new CookieOptions();
option.Expires = DateTime.Now.AddMinutes(30);
Response.Cookies.Append("MyKey", "David", option);

//Read cookie from Request object
string cookieValueFromReq = Request.Cookies["MyKey"];
```

QueryString

- This technique normally used to pass some from the view engine to the controller using the access URL. But it can accept only some limited amount of data which can be passed from one request to another request

```
<h3>Using QueryString</h3>

@Html.ActionLink("Pass Data", "Index","Home", new { Code = 100, Name = "David", Department = "Software" })
```

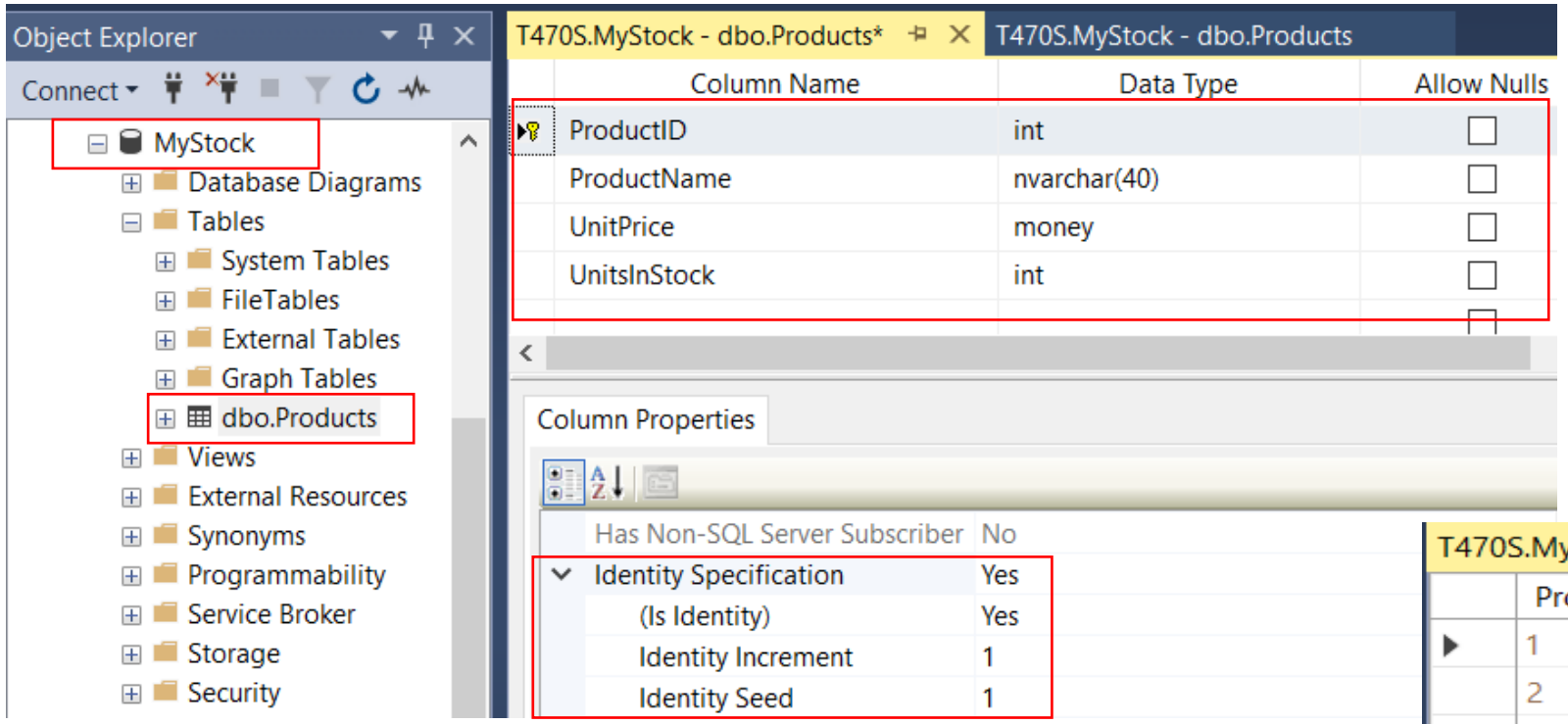
```
public class HomeController : Controller {
    public ActionResult Index(int code, string strName, string strDepartment) {
        string strResult = $"Id ={code}, Name ={strName}, Department ={strDepartment}";
        return Content(strResult);
    }
    public IActionResult Index() {
        return View();
    }
}
```

HttpContext.Items and Cache

- The HttpContext.Items collection is used to store data while processing a single request
- The collection's contents are discarded after a request is processed. The Items collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters
- Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items
- Cached data isn't associated with a specific request, user, or session. Do not cache user-specific data that may be retrieved by other user requests

Demo 03: Working with Entity Framework

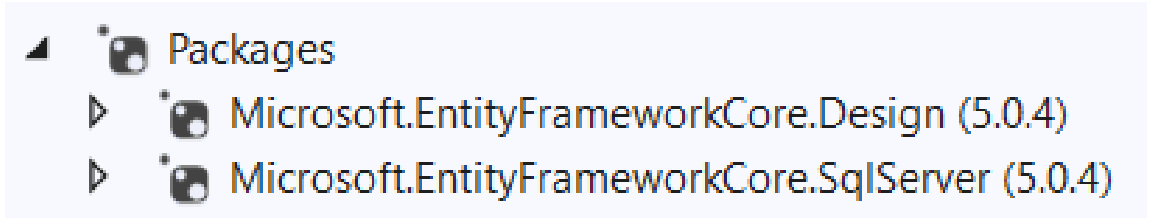
- Create a sample database named **MyStock** for demonstrations



ProductID	ProductNa...	UnitPrice	UnitsInStock
1	Genen Shou...	50.0000	39
2	Alice Mutton	30.0000	17
3	Aniseed Syr...	40.0000	13
4	Perth Pasties	22.0000	53
5	Carnarvon T...	21.3500	0
6	Gula Malacca	25.0000	120
7	Steeleye St...	30.0000	15
8	Chocolade	40.0000	6
9	Mishi Kobe ...	97.0000	29
10	Ikura	31.0000	31

1.Create a ASP.NET Core MVC application named **MyStockApp**

2. Install the following packages from Nuget:



3.Open **appsettings.json ,** click **Add** and add connection string as follows:

appsettings.json

Schema: https://json.schemastore.org/appsettings

```
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "AllowedHosts": "*",
10   "ConnectionStrings": {
11     "MyStockDB": "Server=(local);uid=sa;pwd=123;database=MyStock"
12   }
13 }
```

4.Right-click on the project, select **Open in Terminal.** On **Developer PowerShell** dialog**,** execute the following commands to generate model (Copy and Paste the below command):

```
dotnet ef dbcontext scaffold Name=ConnectionStrings:MyStockDB Microsoft.EntityFrameworkCore.SqlServer --output-dir Models
```

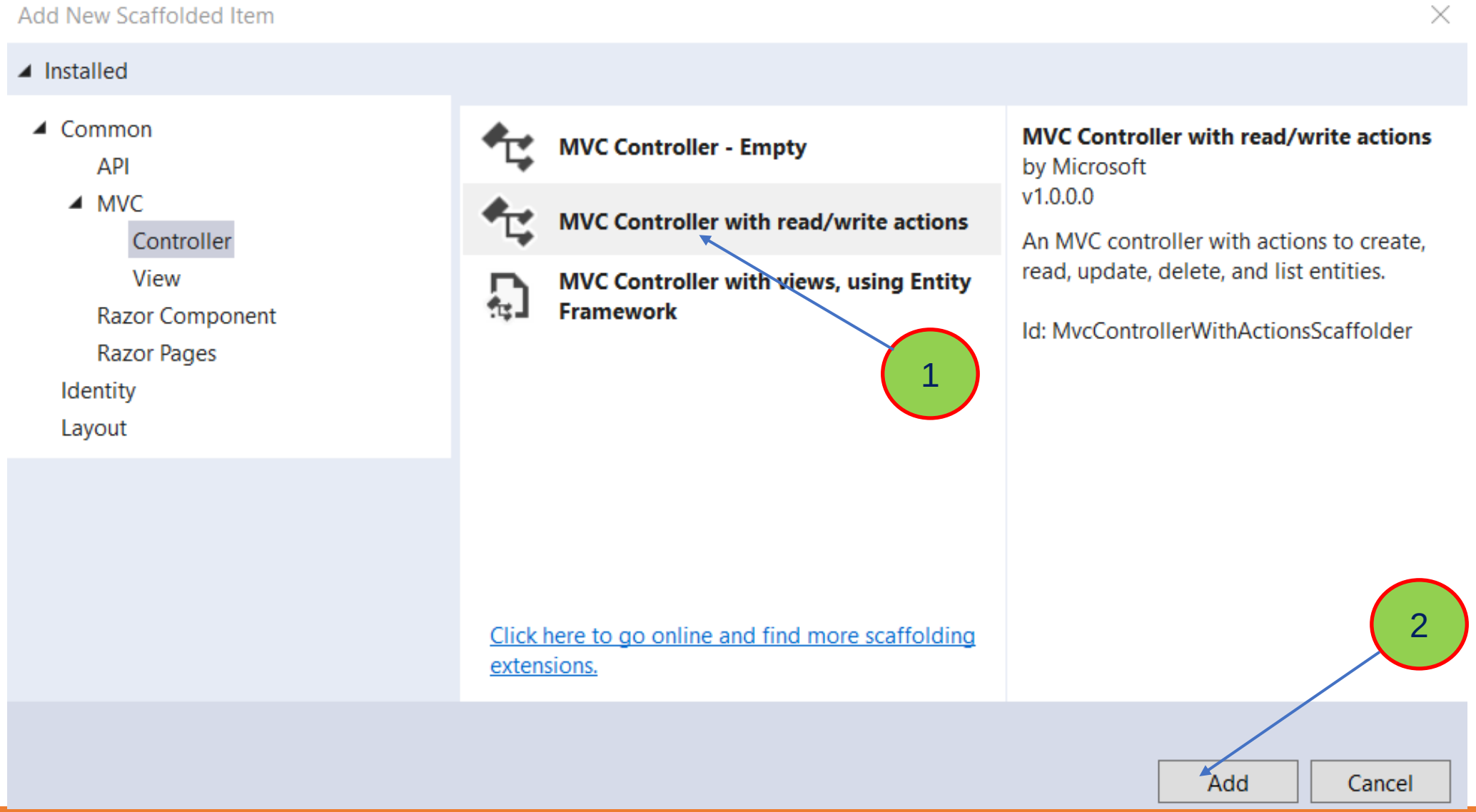
5.Open **Startup.cs** and write codes for **ConfigureService** method as follows:

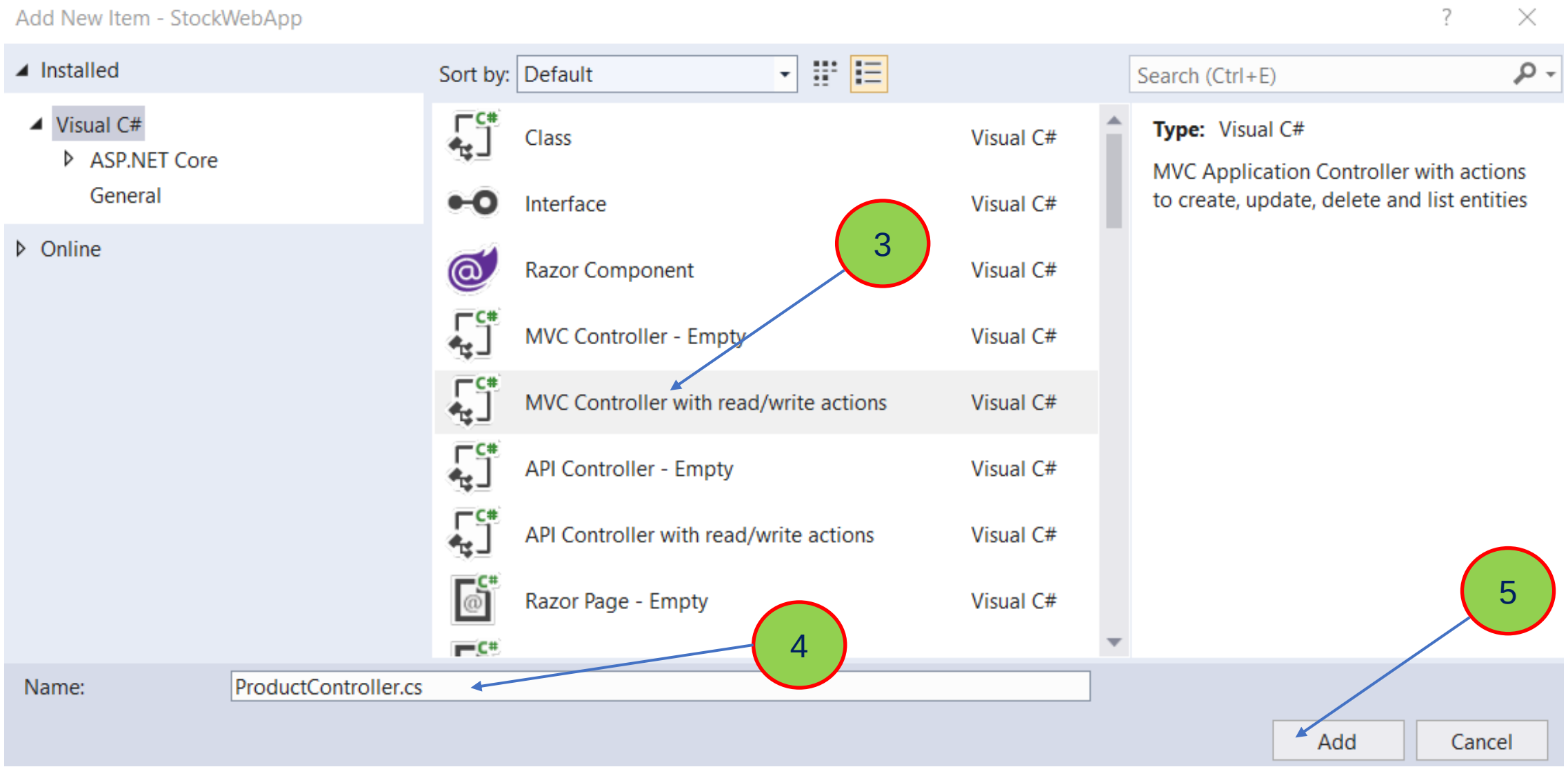
```
//add two namespaces
using StockWebApp.Models;
using Microsoft.EntityFrameworkCore;
//…

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<MyStockContext>(options =>
                        options.UseSqlServer(Configuration.GetConnectionString("MyStockDB")));
        services.AddScoped(typeof(MyStockContext));

        services.AddControllersWithViews();
    }
```

6.Right-click on **Controller** folder | Add | Controller then setup as follows:




Add New Item - StockWebApp
Installed
Visual C#
ASP.NET Core
General
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C#
Interface Visual C#
Razor Component Visual C#
MVC Controller - Empty Visual C#
MVC Controller with read/write actions Visual C#
API Controller - Empty Visual C#
API Controller with read/write actions Visual C#
Razor Page - Empty Visual C#
Type: Visual C#
MVC Application Controller with actions to create, update, delete and list entities
3
4
5
Name: ProductController.cs
Add
Cancel

7.Write codes for action methods of **ProductController.cs** as follows:

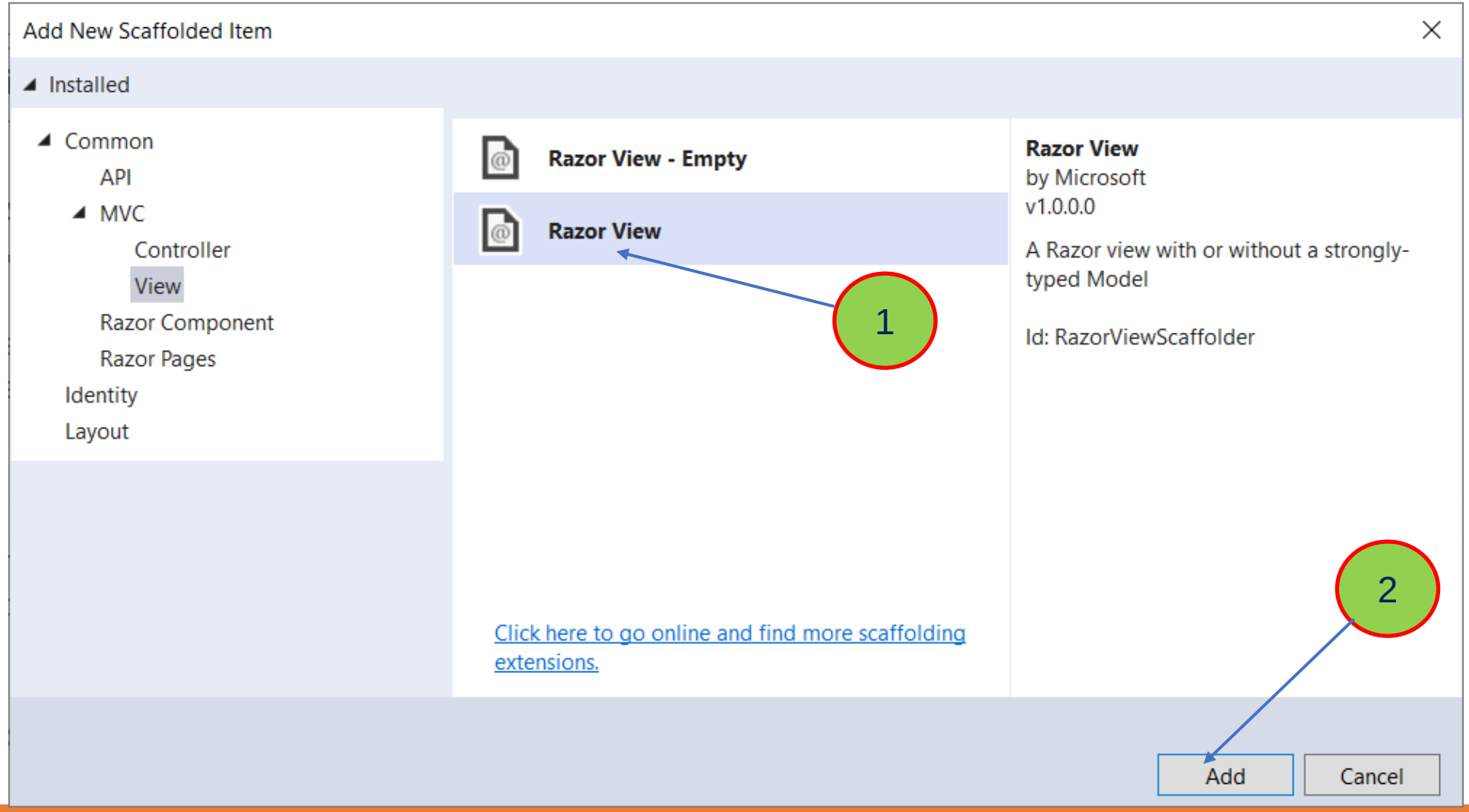
```
//...
using StockWebApp.Models;
using Microsoft.EntityFrameworkCore;

namespace StockWebApp.Controllers{
    public class ProductController : Controller {
        private readonly MyStockContext context;
        public ProductController(MyStockContext context) => this.context = context;
        //Show all Products
        public ActionResult Index(){
            var model = context.Products.ToList();
            return View(model);
        }
        // GET: ProductController/Details/5
        public ActionResult Details(int? id){
            if (id == null){
                return NotFound();
            }
            var product = context.Products.FirstOrDefault(m => m.ProductId == id);
            if (product == null){
                return NotFound();
            }
            return View(product);
        }
```

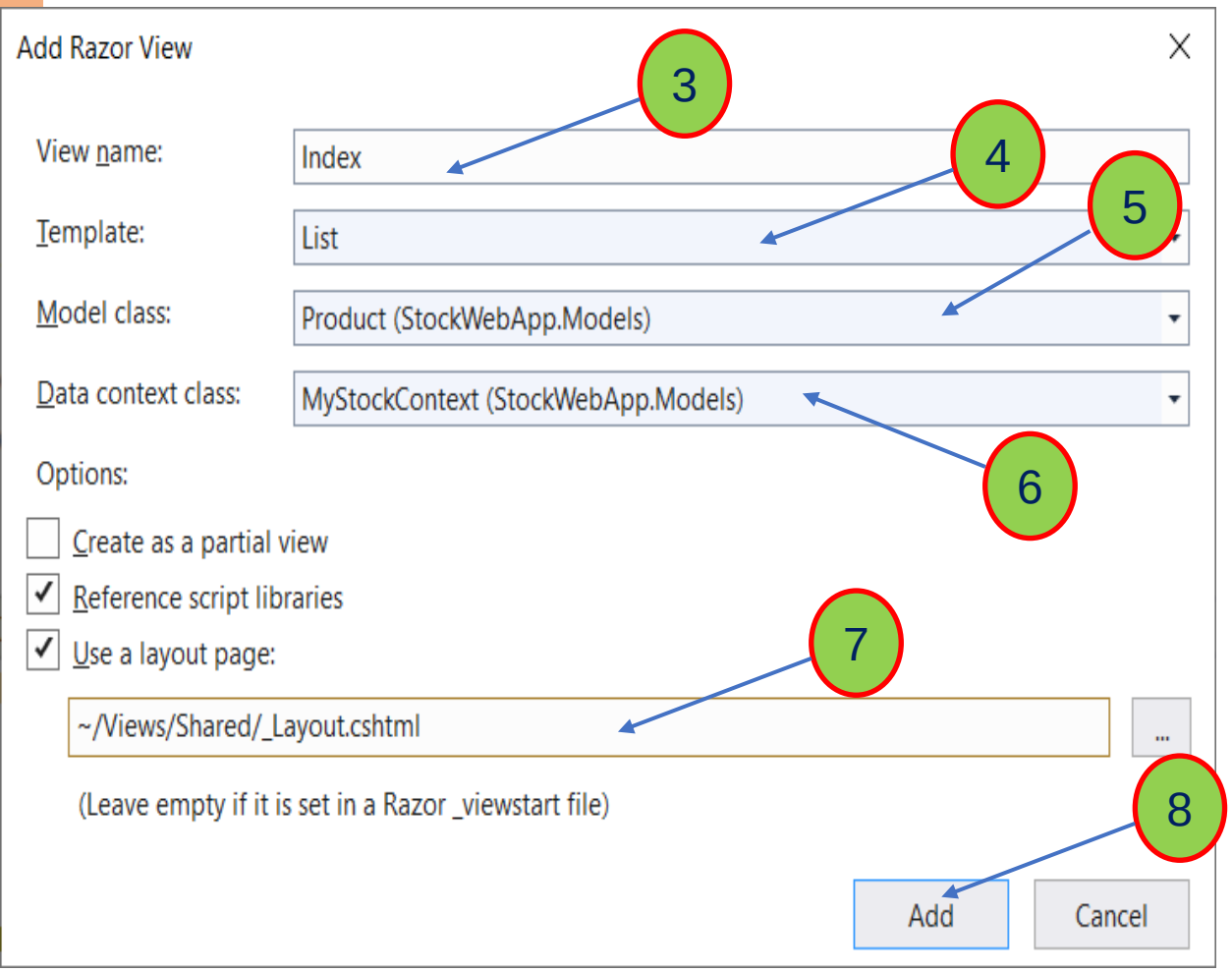
```
// GET: ProductController/Create
public ActionResult Create(){
    return View();
}
// POST: ProductController/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Product product){
    if (ModelState.IsValid){
        context.Add(product);
        context.SaveChanges();
        return RedirectToAction(nameof(Index));
    }
    return View(product);
}
// GET: ProductController/Edit/5
public ActionResult Edit(int? id){
    if (id == null){
        return NotFound();
    }
    var product = context.Products.Find(id);
    if (product == null){
        return NotFound();
    }
    return View(product);
}
// POST: ProductController/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(int? id, Product product){
    if (id != product.ProductId){
        return NotFound();
    }
    if (ModelState.IsValid){
        context.Update(product);
        context.SaveChanges();
        return RedirectToAction(nameof(Index));
    }
    return View(product);
}
// GET: ProductController/Delete/5
public ActionResult Delete(int? id){
    var product = context.Products.Find(id);
    context.Products.Remove(product);
    context.SaveChanges();
    return RedirectToAction(nameof(Index));
}
}//end class
}//end namespace
```

8.Right-click on **View** folder | **Add** | **New Folder** named **Product**

9.Right-click on **Product** folder | **Add** | **View** named **Index** as follows:



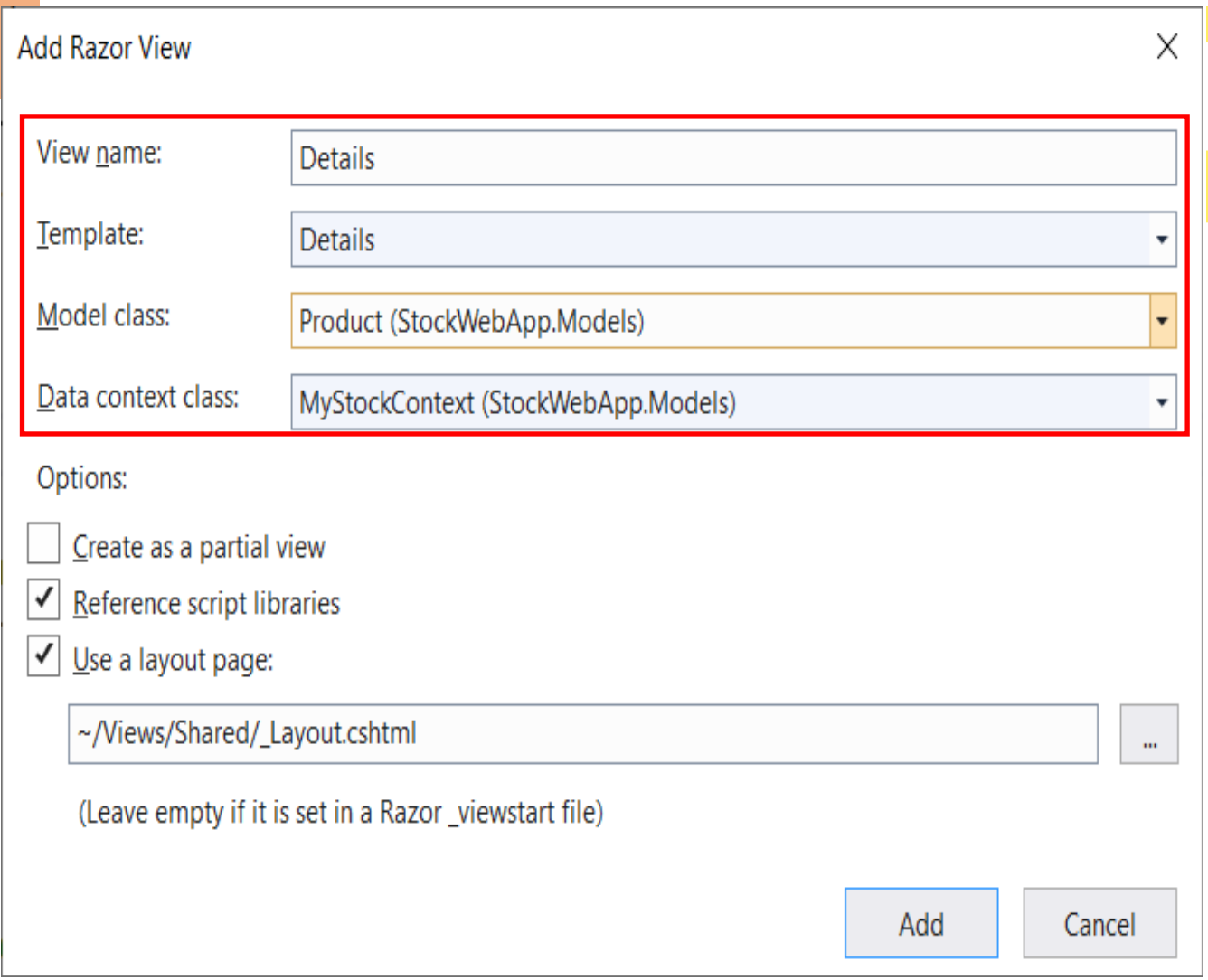
- **Index** view (show product list)




```
@model IEnumerable<StockWebApp.Models.Product>
@{
    ViewData["Title"] = "Product List";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>Product List</h1>
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>...</thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.ProductName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.UnitPrice)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.UnitsInStock)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ProductId">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ProductId">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ProductId">Delete</a>
```

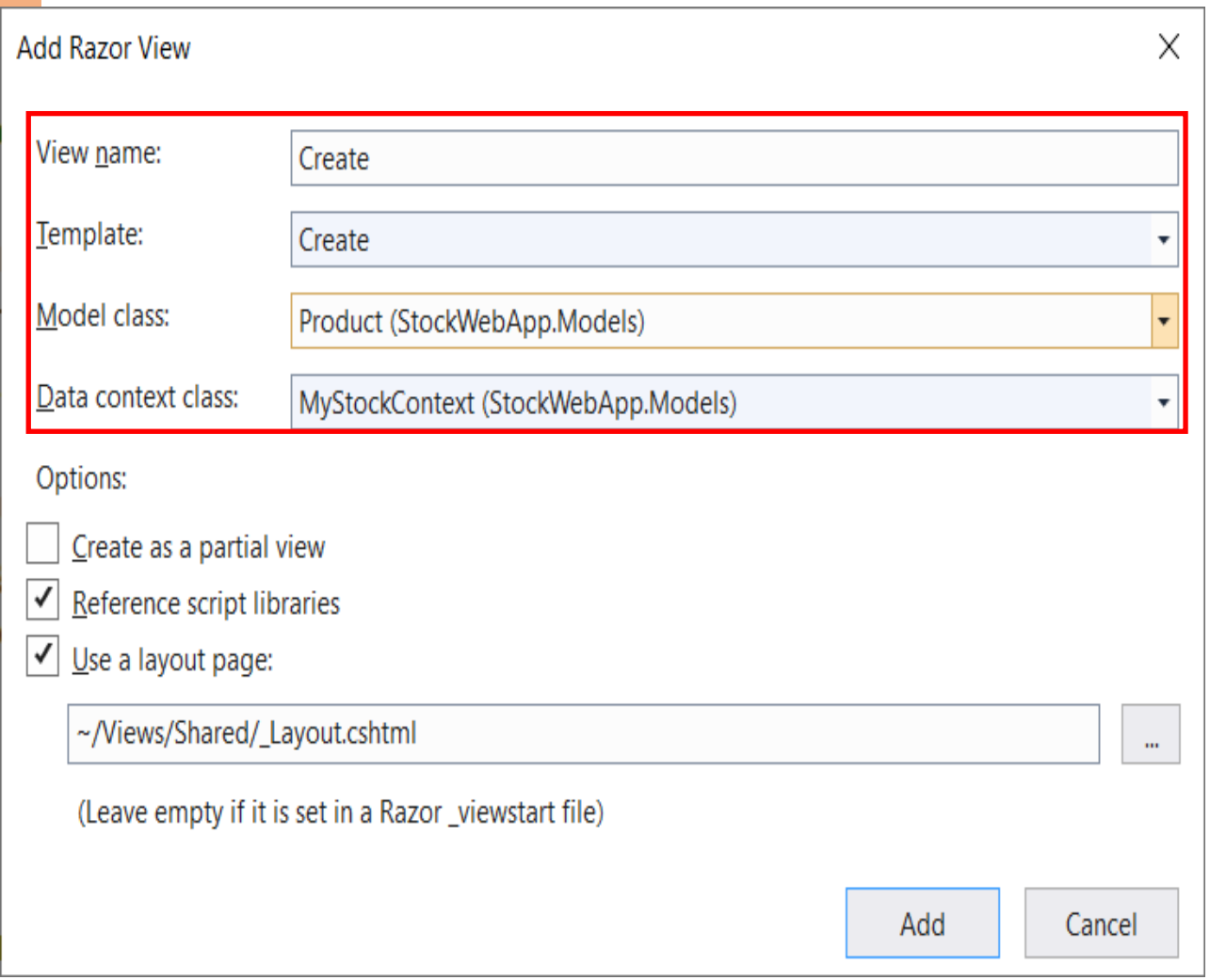
- Repeat this step to add views: **Details**, **Create** and **Edit** as the next figures

Details view



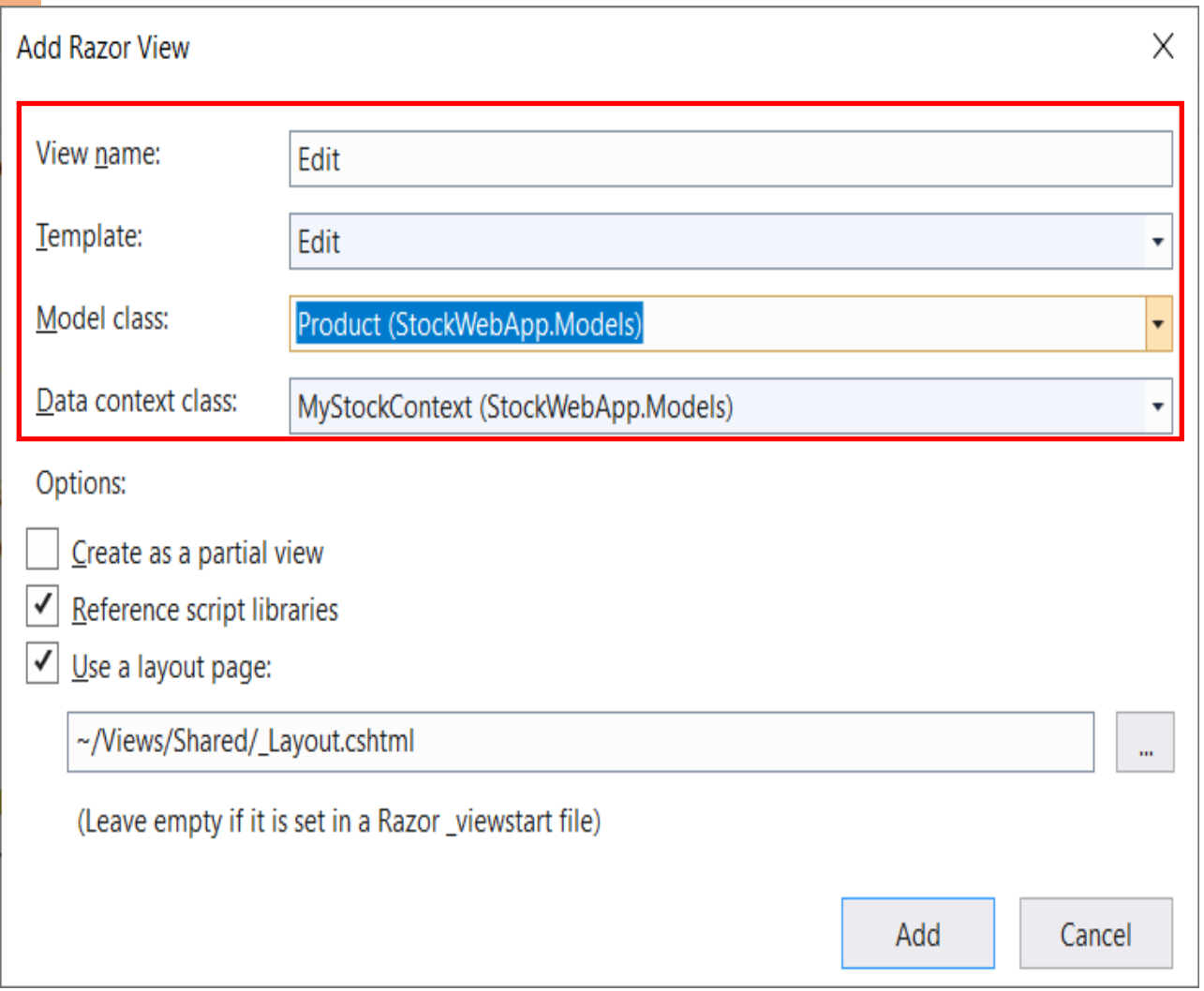

```
@model StockWebApp.Models.Product
@{
    ViewData["Title"] = "Details";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>Details</h1>
<div>
    <h4>Product</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ProductName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ProductName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.UnitPrice)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.UnitPrice)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.UnitsInStock)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.UnitsInStock)
```

Create view




```
@model StockWebApp.Models.Product
@{
    ViewData["Title"] = "Create";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>Create</h1>
<h4>Product</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="ProductName" class="control-label"></label>
                <input asp-for="ProductName" class="form-control" />
                <span asp-validation-for="ProductName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="UnitPrice" class="control-label"></label>
                <input asp-for="UnitPrice" class="form-control" />
                <span asp-validation-for="UnitPrice" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="UnitsInStock" class="control-label"></label>
                <input asp-for="UnitsInStock" class="form-control" />
                <span asp-validation-for="UnitsInStock" class="text-danger"></span>
            </div>
```

Edit view




```
@model StockWebApp.Models.Product
@{
    ViewData["Title"] = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>Edit</h1>
<h4>Product</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="ProductId" />
            <div class="form-group">
                <label asp-for="ProductName" class="control-label"></label>
                <input asp-for="ProductName" class="form-control" />
                <span asp-validation-for="ProductName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="UnitPrice" class="control-label"></label>
                <input asp-for="UnitPrice" class="form-control" />
                <span asp-validation-for="UnitPrice" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="UnitsInStock" class="control-label"></label>
                <input asp-for="UnitsInStock" class="form-control" />
                <span asp-validation-for="UnitsInStock" class="text-danger"></span>
```

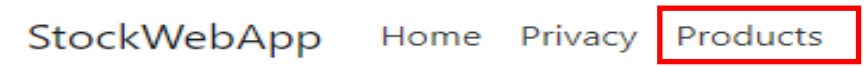
10.Open **index.cshtml** view of **Product** folder and update contents as follows:

```
@{
    ViewData["Title"] = "Product List";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Product List</h1>
```

11.Open **_Layout.cshtml** view in the View | Shared folder, add tags to navigate to **Index** view of **Product** controller as follows then run project:

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Product" asp-action="Index">Products</a>
        </li>
    </ul>
</div>
```



Product List

Create New

ProductName	UnitPrice	UnitsInStock	
Genen Shouyu	50.00	38	Edit | Details | Delete
Alice Mutton	30.00	17	Edit | Details | Delete
Aniseed Syrup	40.00	13	Edit | Details | Delete
Perth Pasties	22.00	53	Edit | Details | Delete
Carnarvon Tigers	21.35	0	Edit | Details | Delete
Gula Malacca	25.00	120	Edit | Details | Delete
Steeleye Stout	30.00	15	Edit | Details | Delete
Chocolade	40.00	6	Edit | Details | Delete
Mishi Kobe Niku	97.00	29	Edit | Details | Delete
Ikura	31.00	32	Edit | Details | Delete

Summary

- Concepts were introduced:
  - Overview ASP.NET Core
  - List the advantages of ASP.NET Core
  - Explain about WebServer
  - Overview ASP.NET MVC Architecture
  - Explain role of the Model, View and Controller
  - Explain about ViewBag, ViewData, TempData and Session
  - Explain about HTML Helper
  - Explain about Model Binding and Model Validation
  - Demo create ASP.NET Core MVC application with Entity Framework