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**. 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 (Model-View-Controller) design pattern 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

The **controller** is the component in the MVC design pattern, who actually **handles the incoming request**. In order to handle the request, 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.

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. **model created by the controller. 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.**

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.

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.

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.

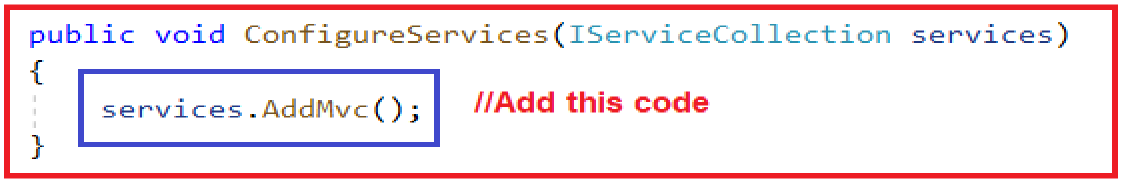
Where MVC is used in the real-time three-layer application?

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.

strongly typed view provides compile-time type checking and intellisense. With intellisense support we can be more productive and the chances of mis-spelling and making typographical errors are almost nil.

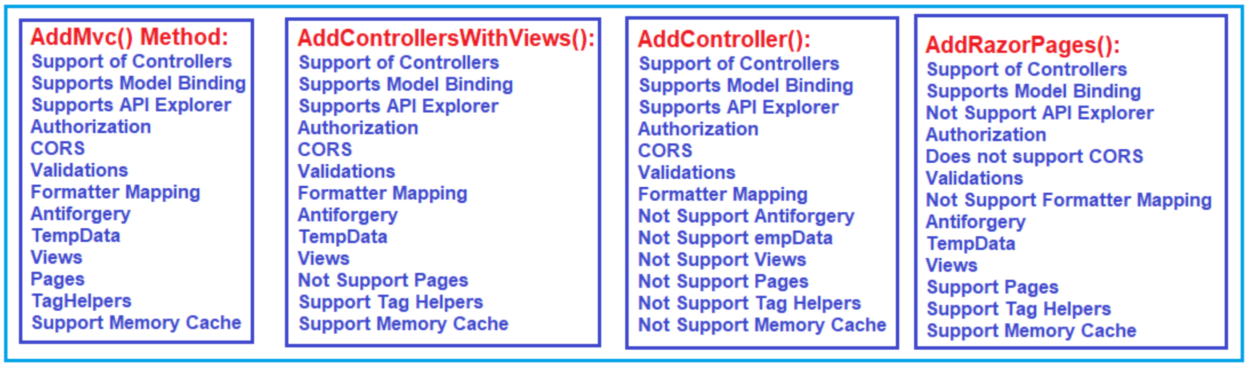
ASP.NET Core MVC- is a lightweight, open-source, highly testable presentation framework that is used for building web apps and Web APIs using the Model-View-Controller (MVC) design pattern. So, the point that you need to remember is, MVC is a design pattern and ASP.NET Core MVC is the framework that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

Set up MVC in ASP.NET Core Application 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/program.cs class



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();     //Add this code
}
```

AddMVC() method, AddControllersWithViews() method.



AddMvc() Method:	AddControllersWithViews():	AddController():	AddRazorPages():
Support of Controllers	Support of Controllers	Support of Controllers	Support of Controllers
Supports Model Binding	Supports Model Binding	Supports Model Binding	Supports Model Binding
Supports API Explorer	Supports API Explorer	Supports API Explorer	Not Support API Explorer
Authorization	Authorization	Authorization	Authorization
CORS	CORS	CORS	Does not support CORS
Validations	Validations	Validations	Validations
Formatter Mapping	Formatter Mapping	Formatter Mapping	Not Support Formatter Mapping
Antiforgery	Antiforgery	Not Support Antiforgery	Antiforgery
TempData	TempData	Not Support empData	TempData
Views	Views	Not Support Views	Views
Pages	Not Support Pages	Not Support Pages	Support Pages
TagHelpers	Support Tag Helpers	Not Support Tag Helpers	Support Tag Helpers
Support Memory Cache	Support Memory Cache	Not Support Memory Cache	Support Memory Cache

Dependency Injection- 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.

ASP.NET Core Framework injects objects of dependency classes through constructor or method by using a built-in IoC (Inversion of Control) 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:

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.

different methods to register a service with Dependency Injection Contains-

The method that we use to register a service will determine the lifetime of that service.

1. Singleton (by default)
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.

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.

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.

An action method can return several types.

1- System.Web.Mvc.ContentResult
2- System.Web.Mvc.EmptyResult
3- System.Web.Mvc.FileResult
4- System.Web.Mvc.HttpStatusCodeResult
5- System.Web.Mvc.JavaScriptResult
6- System.Web.Mvc.JsonResult
7- System.Web.Mvc.RedirectResult
8- System.Web.Mvc.RedirectToRouteResult
9- System.Web.Mvc.ViewResultBase

ViewData- is a dictionary of weakly typed objects which is derived from the ViewDataDictionary class.

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.

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.

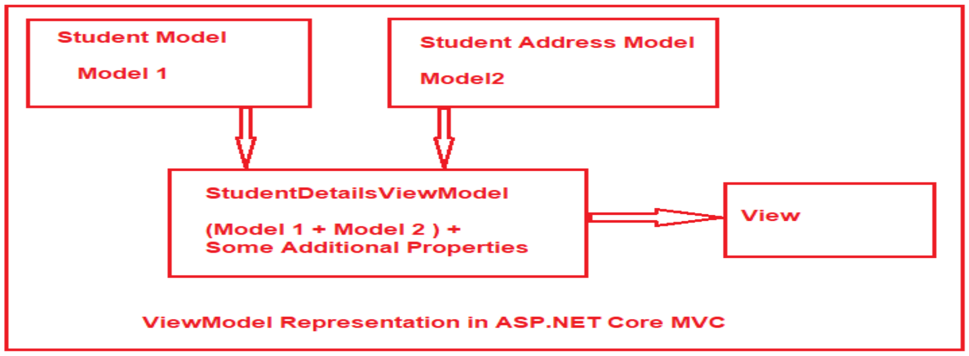
The **ViewData only transfers the data from the controller action method to a view, but not vice-versa. That means it is valid only during the current request.**

<u>ViewBag</u> is a dynamic property of the Controller base class.ViewBag transfers the data from the controller action method to a view only, the reverse is not possible.

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.

<u>ViewModel</u> 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.

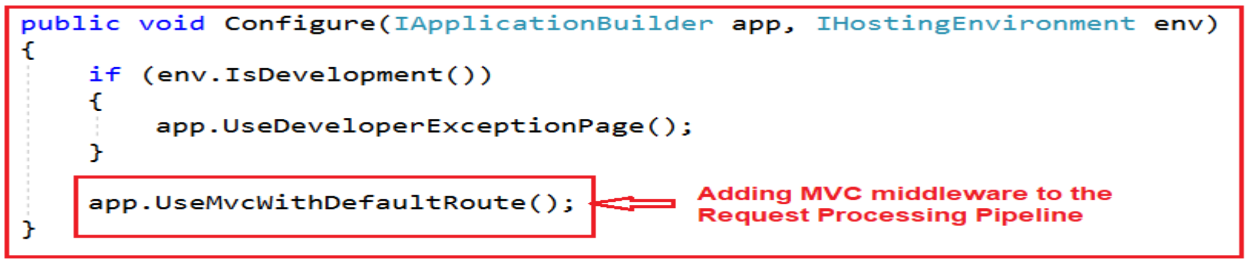


<u>Routing</u> - is a mechanism which is used to inspect the incoming Requests (i.e. URLs) and then mapped that request to the controllers and their action methods. This mapping is done by the routing rules which are defined for the application. We can do this by adding the Routing Middleware to the request processing pipeline.

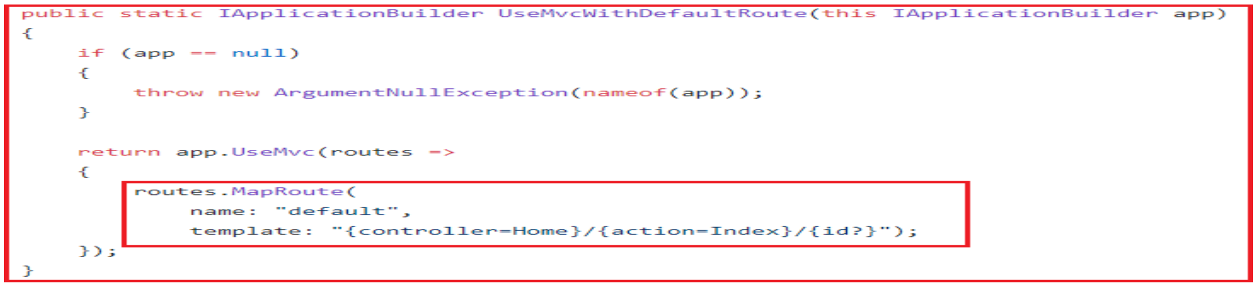
1. Convention Based Routing
2. Attribute-Based Routing.

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.

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.



```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseMvcWithDefaultRoute();
}
```

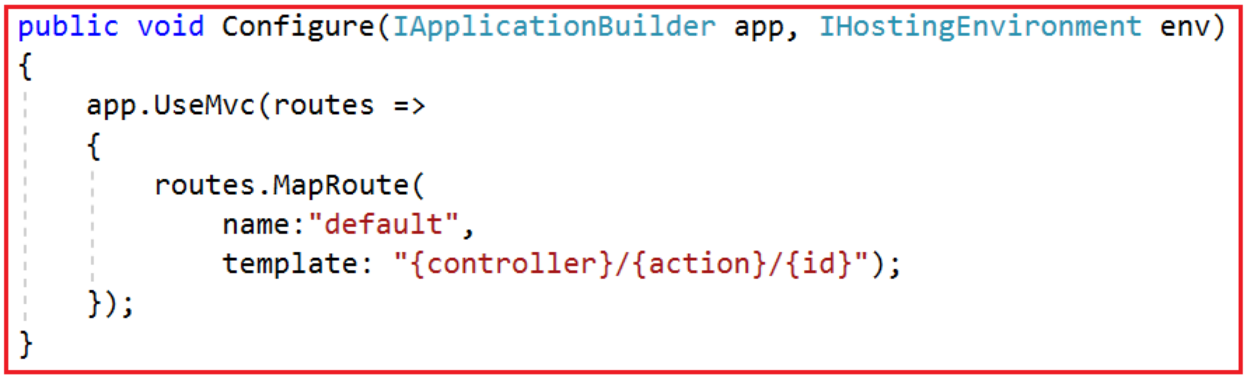


```
public static IApplicationBuilder UseMvcWithDefaultRoute(this IApplicationBuilder app)
{
    if (app == null)
    {
        throw new ArgumentNullException(nameof(app));
    }

    return app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Custom Routing

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.



```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name:"default",
            template: "{controller}/{action}/{id}");
    });
}
```

Attribute Routing 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.

Tokens in Attribute Routing: 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

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.

Route("") attribute to make the Index action method of Home Controller as the default action,

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.

```
[Route("[controller]/[action]")]
```

Attribute Routing vs Conventional Routing :

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.

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. we usually place the layout views in a* **subfolder called "Shared" within the "Views" folder.**

the layout view file is named _Layout.cshtml.

Nowadays, almost all web applications have the following sections. 1- Website Header 2- Website Footer 3- Navigation Menus 4- Main Content Area

_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.*

_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:

@addTagHelper	@tagHelperPrefix	@removeTagHelper	@namespace
@inject	@model	@using	

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag helpers are a new feature and similar to HTML helpers, which help us render HTML., so they are going to be processed on the server to create and render HTML elements in the Razor files.

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.

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Understanding the Anchor Tag Helper 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.

Generating Link using Tag Helper:

<a asp-controller=”Home” asp-action=”Details” asp-route-id=”@student.StudentId”>View</a>

Tag Helpers used to create a Form

1-- Form Tag Helper 2-Input Tag Helper 3- Label Tag Helper 4-Textarea Tag Helper

5- Radio Button Helper 6- Select Tag Helper

Model Binding is a mechanism that 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.

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).

Data Annotation- Data Annotations are Model Binder that is used to perform model validation within a .NET application. The advantage of using the Data Annotation validators is that they enable you to perform validation simply by adding one or more attributes to a class property.

System.ComponentModel.DataAnnotations namespace has a group of classes, attributes, and methods, to make validations.

1. **Required:** Enables you to mark a property as required.
2. **StringLength:** Enables you to specify a maximum length for a string property.
3. **MaxLength:** Enables you to specify the number of maximum and minimum elements in the Array property.
4. **Range:** Enables you to validate whether the value of a property falls between a specified range of values.
5. **RegularExpression:** Enables you to validate whether the value of a property matches a specified regular expression pattern.
6. **Compare:** Enable you to compare the value of two properties
7. **DataType:** Enable you to specify a type to any property like Email, PhoneNumber, Password, CreditCard, PostalCode, Html, Currency, Url, DateTime, etc.

8. **Remote:** Enable you to do remote validation.

LINQ stands for ***Language Integrated Query*** which provides a uniform query syntax in C# and VB.NET to retrieve data from different sources and formats.

1. LINQ is a data querying methodology that provides querying capabilities to .NET languages with a syntax similar to a SQL query.
2. LINQ has great power of querying on any source of data. The data source could be collections of objects, databases or XML files.
3. LINQ was introduced in .NET Framework 3.5 to query the data from different sources of data.
4. The System.LINQ namespace contains set of extension methods that allows us to query the source of data object directly in our code based on the requirement.
5. We can easily retrieve data from any object that implements the IEnumerable<T> interface.
6. LINQ queries return results as objects. It enables you to use an object-oriented approach on the result set and not to worry about transforming different formats of results into objects. You will not get the result of a LINQ query until you execute it.
7. At compile time, LINQ query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification.

Types of LINQ:

There are the following LINQ Objects available in the .Net:

1. LINQ To Objects
2. LINQ To DataSets
3. LINQ To SQL
4. LINQ to XML
5. LINQ To Entities