UNIT – 4

Creating ASP.NET core MVC applications:

.NET Core is a new version of .NET Framework, which is free, open-source, general-purpose development platform maintained by Microsoft. It is a cross-platform framework that runs on Windows, macOS, and Linux operating systems. .NET Core framework can be used to build different types of applications such as mobile, desktop, web, cloud, IoT, machine learning, game etc. ASP.NET Core is the new version of the ASP.NET web framework mainly targeted to run on .NET Core platform.

Why .NET Core?

→ Supports multiple platforms.
→ Fast.
→ Integration with Modern UI Frameworks like Bootstrap, ReactJS etc.
→ Contains IoC Container which makes it maintainable and testable.
→ Can be hosted on multiple platforms with any server such as IIS, Apache etc.
→ ASP.NET Core application runs on .NET Core, which is smaller than the full .NET Framework.

⊛Setting Up The Environment:

ASP.NET Core – wwwroot Folder: By default, the wwwroot folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root. There should be seperate folders for different types of static files such as JavaScript, CSS, Images etc. We can access static files with base URL and file name.

For example: http://localhost:<port>/css/app.css

Rename wwwroot Folder: We can rename wwwroot folder to any other name as per our choice and set it as a web root while preparing hosting environment in the program.cs.

ASP.NET Core – Startup Class: ASP.NET Core application must include Startup class. As the name suggests, it is executed when the application starts. The startup class can be configured at the time of configuring the host in the Main() method of Program class. Startup class includes two methods ConfigureServices() and Configure().

ConfigureServices(): The ConfigureServices method is a place where we can register our dependent classes with the built-in IoC container. After registering dependent class, it can be used anywhere in the application. We must need to include it in the parameter of the constructor of a class where we want to use it. The IoC controller will inject it automatically.

Configure(): The Configure method is a place where we can configure application request pipeline for our application using IApplicationBuilder instance that is provided by the built-in IoC container.

⊛ Controller: Controllers are the central unit of our ASP.NET MVC application that will combine our Model with a View and serve the result to the end-user. The controller can act on its own for the basic operations, e.g, delivering a simple text message or redirecting user to somewhere else. Actually, controlls are controlling the overall flow of the application taking the input and rendering the proper output.

Where are controllers placed?

⇒ Controllers are usually placed in a folder called "Controllers", directly in the root of our MVC project. They are usually named based on their purpose, with the word "Controller" as a suffix.

Controller Responsibilities:

→ The controller is responsible to Handle Request from the User.

→ The Controller Action method executes the application logic and builds a model.

→ Finally it should return the Result in HTML/File/Json/XML or whatever the format requested by the user.

Creating/Adding a Controller:

Controllers can be created by selecting Controllers folder and right-click and select the controller option. There are two main options:

- MVC Controller
- API Controller

Both the MVC and API Controller inherits from the same Controller class and there is not much difference between them, except that API Controller is expected to return the data in serialized format to the client.

Further, we have three options under both types of controllers.

- Empty
- With Read/Write Actions
- With Views, using entity framework.

The controller class must satisfy at least one of the following conditions:

→ The class name is suffixed with "Controller".
→ The class inherits from a class whose name is suffixed with "Controller".
→ The class is decorated with the [Controller] attribute.

⊛. Actions:

Each public method in a controller is known as action method, meaning we can invoke it from the Web via some URL to perform an action. For this, we use concept called **Routing**. Routing is responsible for matching incomming HTTP requests and dispatching those requests to action methods on our controller.

Edit action can only be accessed with a GET request. This has the added benefit of allowing us to have multiple methods with the same name, as long as they don't accept the same request method. So for instance, we could have two methods called Edit: The first one would be available for GET requests and generate the FORM for editing an item, while the second one would only be available for POST requests and be used to update the item when the FORM was posted back to the server.

Action Result Types:

Actions are the methods defined on Controller. Actions are invoked when a requested URL is matched to an Action on a Controller. When the action finishes it's work, it will usually return something to the client and that something will usually be implementing the IActionResult interface.

Following are some methods for generating an Action result:

Content(): returns the specified string as plain text to client.

View(): returns a View to the client.

PartialView(): returns a Partial View to the client.

File(): returns the content of a specified file to the client.

Json(): returns a JSON response to the client.

StatusCode(): returns a custom status code to the client.

⊛ Rendering HTML with Views:

In the MVC pattern, the view handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are .cshtml files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in Views folder at the root of the app.

The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the About.cshtml, Contact.cshtml, Index.cshtml etc. webpages. When a user requests one of these three webpages, controller actions in the Home controller determine which of the three views is used to build and return a webpage to the user.

Q. How do you render HTML with views? Explain.

Ans:- Write all the above theory and example given at last of this topic.

Benefits of using views:

Views help to establish separation of concerns (SoC) within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes our app modular, which provides several benefits:

→ The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.

→ The parts of the app are loosely coupled. We can build and update the app's views separately from the business logic and data access components.

→ It's easier to test the user interface parts of the app because the views are seperate units.

→ Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a View:

Views that are specific to a controller are created in the Views [ControllerName] folder. Views that are shared among controllers are placed in the Views/Shared folder. To create a view, add a new file and give it the same name as it's associated controller action with the .cshtml file extension. To create a view that corresponds with the About action in Home controller, create an About.cshtml file in the Views/Home folder:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"].</h3>
<p>Use this area to provide additional information.</p>
```

⊛ Razor Syntax:

Razor is a view engine for MVC framework. Razor is a markup syntax that lets us embed server-based code into web pages. Server based code can create dynamic web content on the fly, while a webpage is written to the browser. It has the power

of traditional ASP.NET (.ASPX) markup, but it is easier to use, flexible and lightweight. Razor markup starts with the @ symbol within html document.

Example: Write necessary html and C# code in razor page to display all the numbers from 1 to 100. All the even numbers must be displayed in green color and odd numbers must be displayed in red color.

Solution:

```
<html>
<body>
    @for (int i=0; i<100; i++)
    {
        if (i%2 ==0)
        {
            <h2 style="color: green">@i </h2>
        }
        else
        {
            <h2 style="color: red">@i </h2>
        }
</body>
</html>
```

⊛ URL Routing:

URL Routing is a mechanism in which it will inspect the incoming Requests (i.e, URLs) and then maps 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 middleware to the request processing pipeline.

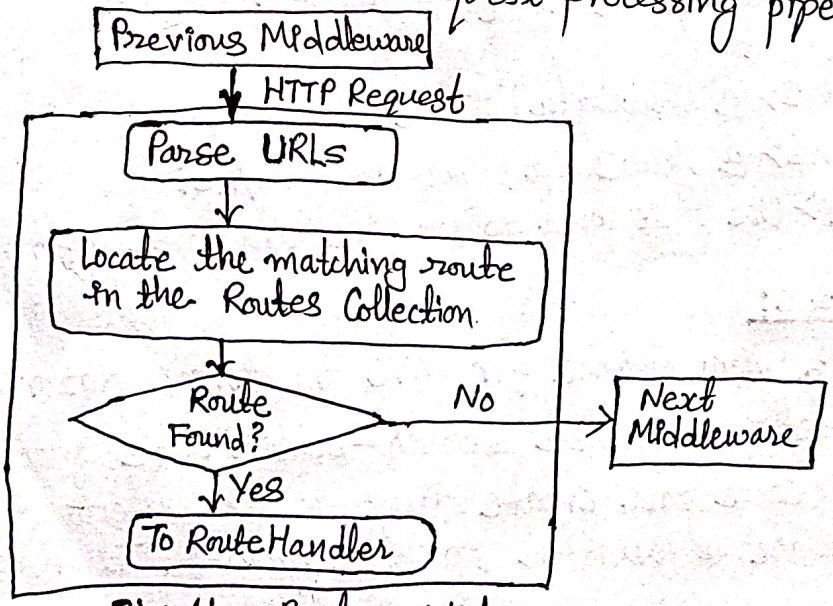


Fig: How Routing Works.

When the Request arrives at the Routing Middleware it does the following:
→ It Parses the URL
→ Searches for matching Route in the Route Collection.
→ If the Route found, then it passes the control to RouteHandler.
→ If Route not found, it gives up and invokes the next Middleware.

i) Convention 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 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.

ii) Attribute-Based Routing: 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. With the help of ASP.NET Core Attribute Routing, we can use the Route attribute to define routes for our application.

⊛ Understanding Tag Helpers:

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

There are many built-in Tag helpers for common tasks, such as creating forms, links, loading assets etc. Tag Helpers are authored in C#, and they target HTML elements based on the element name, the attribute name, or the parent tag. For example, the built-in LabelTagHelper can target the HTML <label> element when LabelTagHelper attributes are applied.

Managing Tag Helper scope:

Tag Helpers scope is controlled by a combination of @addTagHelper, @removeTagHelper, and the "!" opt-out character.

i) @addTagHelper: The @addTagHelper directive makes Tag Helpers available to the view. In this case, the view file is Pages/_ViewImports.cshtml, which by default is inherited by all files in the Pages folder and

subfolders; making Tag Helpers available. The '*' after @addTagHelper specify that all Tag Helpers in the specified assembly (Microsoft.AspNetCore.MVC.TagHelpers) will be available to every view file in the Views directory or subdirectory.

To expose all of the Tag Helpers in project, we would use the following:

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

ii) @removeTagHelper: The @removeTagHelper has the same two parameters as @addTagHelper, and it removes a Tag Helper that was previously added. For example, @removeTagHelper applied to a specific view removes the specified Tag Helper from the view. Using @removeTagHelper in a Views/Folder/_ViewImports.cshtml file removes the specified Tag Helper from all of the views in Folder.

iii) Opt-out: We can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, Email validation is disabled in the with the Tag Helper opt-out character.

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

List of Some Built-in Tag Helpers:

Tag Helper	Targets	Attributes
Form Tag Helper	<form>	asp-action, asp-all-route-data, asp-area, asp-controller, asp-host, asp-page, asp-protocol, asp-handler
Anchor Tag Helper	<a>	asp-action, asp-all-route-data, asp-controller, asp-host, asp-page, asp-handler.
Cache Tag Helper	<cache>	enabled 1, expires-after 2, expires-on 3, priority 5
Image Tag Helper	<img>	append-version.
Input Tag Helper	<input>	for
Label Tag Helper	<label>	for

⊛. HTML Helper:

An HTML helper is a method that is used to render html content in a view. HTML helpers are implemented as extension methods. HTML helpers will greately reduce the amount of HTML that we have to write in a view. Views should be as simple as possible.

For Example: To produce the HTML for a textbox with id="firstname" and name="firstname", we can type all the html in the view as:

```
<input type="text" name="firstname" id="firstname" />
```

OR we can use the "TextBox" html helper as:

```
@Html.TextBox("firstname").
```

To set a value, along with name:

```
@Html.TextBox("firstname", "John")
```

This html helper generates the following HTML:

```
<input id="firstname" name="firstname" type="text" value="John" />
```

Extension Method	Html Control
Html.ActionLink()	<a>
Html.TextBox()	<input type="textbox">
Html.TextArea()	<input type="textarea">
Html.CheckBox()	<input type="checkbox">
Html.Password()	<input type="password">
Html.Label()	<label>

Example: Generating Links using Tag Helpers:

Let's say we want to view a specific employee details. The number 5 is the ID of the employee whose details we want to view, and we want to generate hyperlink. Now, there are many possible we could do this in razor view:

Normal HTML: `<a href="/home/details?id=5"> View </a>`

Using HTML Helper: `@Html.ActionLink("View", "home/details", new {id=5})`

Using Tag Helper:

```
<a asp-controller="home" asp-action="details"
   asp-route-id="5"> View </a>.
```

⊛. Model:

The model is a collection of objects, which hold the data of our our application and it may contain the associated business logic. The model classes represent domain-specific data and business logic in the MVC application and are simply C# classes. In the ASP.NET MVC application, all the Model classes must be created in the Model folder. The model is divided into several categories based on how and where they are used. The three main distinctions are:-

1) Domain Model: A Domain Model represents the object that represents the data in database. The Domain Model usually has one to one relationship with the tables in the database.

2) View Model: The View Model refers to the objects which hold the data that needs to be shown to the user. The View Model is related to the presentation layer of our application.

3) Edit Model: The Edit Model or Input Model represents the data that needs to be presented to the user for modification/inserting. The UI Requirement of Product for Editing can be different from the model required for Viewing.

⊛. Model Binding:

The Model Binding 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.

Routing binds the values from HTTP request to acton's parameter, using model binder. The model binding system:

→ Retrives data from various sources such as route data, form fields, and query strings.

→ Provides the data to controllers and Razor pages in method parameters and public properties.

Following are the data sources in the order, which model binding looks through:

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.

Model Binding in ASP.NET Core with Complex/User Defined Type:

Model Binding with ASP.Net Core works with Complex type or user defined type like object. If the request is complex, i.e, we pass data in request body as an entity with the desired content-type, then such kind of request is mapped by Complex model binder. The Following example demonstrates the concept:

1. Create View Model named StudentModel.cs in the Model Directory.

```
namespace WebApplication57.Models
{
    public class StudentModel
    {
        public string fullname {get; set;}
        public int roll {get; set;}
        public string email {get; set;}
    }
}
```

2. Now write a strongly typed html view that make use of above model.

```
@model WebApplication57.Models.StudentModel
<form asp-action="/submit" method="post">
    <div>
        <label asp-for="fullname"></label>
        <input asp-for="fullname"/>
    </div>

    <div>
        <label asp-for="roll"></label>
        <input asp-for="roll"/>
    </div>

    <div>
        <label asp-for="email"></label>
        <input asp-for="email"/>
    </div>
    <div> <input type="submit" value="Submit"/> </div>
</form>
```

3. Now write the necessary action methods in the Home Controller one for display the above view with one student record, and another to display the updated field values by mapping the complex type StudentViewModel Form View to Action Method.

```
using Microsoft.AspNetCore.Mvc;
using WebApplication57.Models;
namespace WebApplication58.Controllers
{
    public class HomeController : Controller
    {
        [HttpGet]
        public IActionResult Index()
        {
            StudentModel s=new StudentModel(){fullname="Ram Bdr Thapa",
              roll=5, email="a@b.com"};
            return View(s);
        }
        [HttpPost]
        public IActionResult Submit (StudentModel s)
        {
            return Content("Full Name="+s.fullname +"Roll="+s.roll
                  + "Email="+s.email);
        }
    }
}
```

When the user presses the submit button it will call Submit (StudentViewModel s) action method such that the updated model state in view is mapped to the parameters.

⑧. Model Validation:

Model Validation is a process to ensure the data received from the View is appropriate to bind the Model. If it is not, then appropriate error messages are displayed on the View, and that will help user to rectify the problem. ASP.NET MVC include several built-in-attribute classes in the System.ComponentModel.DataAnnotations for this purpose. Data Annotations are used to add metadata for

ASP.NET MVC and ASP.NET data controls. We can apply these attributes to the properties of the model class to validate and display appropriate validation messages to the users.
Some of these attributes includes:

1). [Required]: Specifies that a property value is required. This validates that the field is not null.
Example: [Required(ErrorMessage="Full name must be provided.")]

2). [StringLength]: Specifies the minimum and maximum length of characters that are allowed in a string type property.
Example: [StringLength(50, MinimumLength=5, ErrorMessage="Full name must be within 5 to 50 characters")]

3). [Range]: Specifies the numeric range constraints for the value of a property.
Example: [Range(1, 100, ErrorMessage="Number must be in range 1 to 100")].

4) [Compare]: This attribute validates that two properties in model class match like password and compare password.
Example: [Compare("Password", ErrorMessage="Password and confirm password do not match.")].

5. [EmailAddress]: This validates the property has email address format.
Example: [EmailAddress(ErrorMessage="Invalid Email Address.")]

6. [RegularExpression]: Specifies that a property value must match the specified regular expression. This validates that the property value matches a specified regular expression.
Example: [RegularExpression("[a-z]{2}-[0-9]{4}-[0-9]{4}", ErrorMessage="Data must be in format xx-0000-0000")].

Example: Create View Model named StudentModel.cs in the Model Directory with necessary validation rule and validation message.

```
using System.ComponentModel.DataAnnotations;
namespace WebApplication59.Models
{ public class StudentViewModel
  {
    [Display(Name="Full Name")]
    [Required(ErrorMessage="Full Name must be provided")].
    [StringLength(50, MinimumLength=5, ErrorMessage="Full Name must
     be within 5 to 50 characters")].
    public string fullname {get; set;}

    [Range(1,100, ErrorMessage="Roll no. must be in range 1 to 100")]
    [Required(ErrorMessage="Roll must be provided.")]
    [Display(Name="Roll Number")]
     public int roll {get; set;}

    [Required(ErrorMessage="Email Address must be provided")]
    [Display(Name="Email Address")]
    [EmailAddress(ErrorMessage="Provide valid email address")].
     public string email {get; set;}
    }
}
```

Custom Validation with Validation Attribute:

Custom validation is useful when the inbuilt validation attribute doesn't solve our validation problem. To define custom validation attribute and work it similar to that of built in attribute, we have to create a class and inherit it from the Validation Attribute class. After that, we can simply override IsValid() method and write our own validation logic.

Example: Let's create a validation attribute that let us define a Data type of field that only accepts a date earlier than today's date.

Steps:

1. Create Validation class code.

```
using System.ComponentModel.DataAnnotations;
public class DateValidationAttribute : ValidationAttribute
{
    public override bool IsValid (object value)
    {
        DateTime todayDate = Convert.ToDateTime (value);
        if (todayDate <= DateTime.Now)
        { return (true);
        }
        else
        { return (false);
        }
    }
}
```

2. To use this validation attribute, we write as an attribute to the model property as we write other in-built validations in the model class code.

```
[DateValidation]
public DateTime BirthDate {get; set;}
[DateValidation (ErrorMessage = "Sorry, the date can't be later
                 than today's date")]
public DateTime MyBirthDate {get; set;}
```

⊛. Web API Applications:

1) API Controllers: It is the most important part of creating a Controller for the Web API. This controller is just a normal controller, that allows data in the model to be retrived or modified, and then deliver it to the client. It does this without having to use the actions provided by the regular controllers.

The data delivery is done by following a pattern known by name as **REST**. REST stands for REpresentational State Transfer pattern, which contains 2 things:

1) Action methods which do specific operations and then deliver some data to the client. These methods are decorated with attributes that makes them to be invoked only by HTTP requests.

1) URLs which defines operational tasks. These operations can be- sending full or part of a data, adding, deleting or updating records.

2) JSON: The new built-in JSON support is System.Text.Json. The System.Text.Json namespace provides high-performance, low-allocating, and standards-compliant capabilities to process JSON. It also provides types to read and write JSON text encoded as UTF-8, and to create an in-memory document object model (DOM) for random access of the JSON elements within a structured view of data. JSONPatch is a method of updating documents on an API in a very explicit way.

Adding JSON Patch to our ASP.Net Core Project:

Inside Visual Studio, we should run Package Manager console to install the official JSON Patch library.

```
Install-Package Microsoft.AspNetCore.JsonPatch.
```

Then we can update as in the example below:

```
[Route ("api/[controller]")]
public class PersonController:Controller {
    private readonly Person _defaultPerson = new Person
    {
        FirstName = "Jim",
        LastName = "Smith"
    };
    [HttpPatch ("update")]
    public PersonPatch([FromBody] JsonPatchDocument<Person> personPatch)
    {
        personPatch.ApplyTo(_defaultPerson);
        return _defaultPerson;
    }
}
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

When we call this endpoint with the following payload:

```
[
    {"op": "replace", "patch": "FirstName", "value": "Bob"}
]
```

We get response of:

```
{ "FirstName": "Bob",
  "LastName": "Smith"
}
```

Now our FirstName "Jim" got changed to "Bob" by using JSONPatch.

3) Dependency Injection (DI):

Dependency Injection is a technique in which an object receives other objects that it depends on. These other objects are called dependencies. Generally the receiving object is called a client and the passed or injected object is called a service.

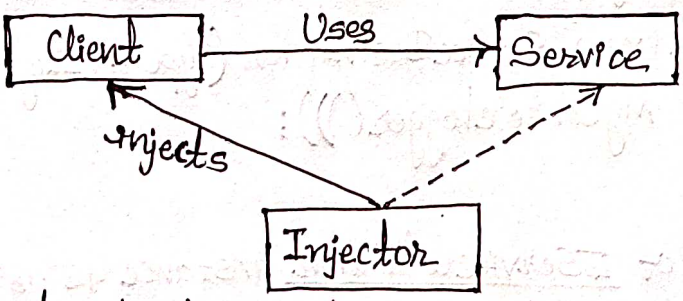
Dependency injection involves four roles:

i) the service object(s) to be used.

ii) the client object object that is depending on the service(s) it uses.

iii) the interfaces that define how the client may use the services.

iv) the injector, which is responsible for constructing the services and injecting them into the client.



The intent behind dependency injection is to achieve seperation of concerns. This can increase readability and code reuse. The injector class injects dependencies broadly in different ways, one among them is through constructor.

4) IoC Container:

ASP.NET Core is designed from scratch to support Dependency Injection. ASP.NET Core injects objects of dependency classes through constructor by default by using built-in IoC container. The built-in container is represented by IServiceProvider implementation that supports constructor injection by default. The types (classes) managed

by built-in IoC container is called services. In order to let the IoC container automatically inject our application services, we first need to register them with IoC container.

Registering Application Service:

Consider the following example of simple ILog interface and its implementation class.

```
public interface ILog
{
    void Info(string str);
}
class MyConsoleLogger : ILog
{
    public void Info(string str)
    {
        Debug.WriteLine(str);
    }
}
```

Let's register above ILog with IoC container in ConfigureServices() method as shown below:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.Add(new ServiceDescriptor(typeof(ILog), new
                                MyConsoleLogger()));
    }
}
```

Here, Add() method of IServiceCollection instance is used to register a service with an IoC container. The ServiceDescriptor is used to specify a service type and it's instance. We have specified ILog as service type and MyConsoleLogger as it's instance.

#The built-in IoC container supports three kinds of lifetimes:

1) Singleton: IoC container will create and share a single instance of a service throughout the applications lifetime. Singleton lifetime services are created the first time they are requested and then every subsequent request will use the same instance.

Transient: The IoC container will create a new instance of the specified service type every time we ask for it. Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

Scoped: IoC container will create an instance of the specified service type once per request and will be shared in a single request. Scoped objects are the same within a request, but different across different requests.

Example: The following example shows how to register a service with different lifetimes.

```
public void ConfigureServices (IServiceCollection services)
{ //singleton
  services.Add (new ServiceDescriptor (typeof (ILog), new MyConsoleLogger()));

  //Transient
  services.Add(new ServiceDescriptor (typeof (ILog), typeof (MyConsoleLogger),
      ServiceLifetime.Transient));

  //Scoped
  services.Add (new ServiceDescriptor (typeof (ILog), typeof (MyConsoleLogger),
      ServiceLifetime.Scoped));
```

Constructor Injection:

Once we register a service, the IoC container automatically performs constructor injection if a service type is included as a parameter in a constructor. For example, we can use ILog service type in any MVC controller. Consider the following example.

```
public class HomeController: Controller
{ ILog _log;
  public HomeController (ILog log) //Constructor Injection
  { _log = log;
  }
  public IActionResult Index()
  { _log.Info("Executing /home/index");
    return View();
  }
}
```

In the above example, an IoC container will automatically pass an instance of MyConsoleLogger to the constructor of HomeController. We don't need to do anything else. An IoC container will create and dispose an instance of ILog based on the registered lifetime.

⊛. Rendering with HTML with Views [Example]:

→ पहिलो यो topic मा example लेख्न दुईद, so last end of this topic मा example थपेको

The following program shows how a view is displayed.

1. HTML View Defined as /Views/Home/Index in file Index.cshtml.

```
<h1>This is Index Page</h1>
```

2. Action Method Index() defined at Controller/Home in file HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace MyMvcApplication.Controller
{
 public class HomeController:Controller
 {
  public IActionResult Index()
  { return View();
  }
 }
}
```