ASP.NET Core 6

Introduction:

* API: Application programming Interface.
* Application: It tells what software has the program being run?
* Program: What task is being done/what does the program do?
* Interface: From where are you telling the program to run?
* Rest: Representational State Transfer. It is used to complete a Web Service (API) using HTTP.
* Authentication: Proving your Identity
* Authorization: Limited access
* Dot net core is a free open source and cross platform, created for building modern cloud-based applications and every .NET developer feels proud of it.
* In the year 2016, Microsoft has come up with a new revolution, Microsoft.NET Core 1.0.

Open-source Means:

* In general, open source refers to any program whose source code is made available for use or modification as users or other developers see fit. Open-source software is usually developed as a public collaboration and made freely available.

ASP.NET Core features:

* Cross platform, Open source now runs our app over Linux, Windows, Mac.
* Fast Development- fast work over the Browsers
* Work in your editors- now we can work not only in visual studio. We can choose Visual Studio Code.
* It is based on Model-view-Controller (MVC).

What is ASP.NET Core?

* ASP.NET Core is a new version of ASP.NET developed by Microsoft.
* It is open-source framework for developing we applications and it can be run on Windows, Mac, Linux.
* It has been completely rewritten from scratch and it was initially launched as ASP.NET 5 but then was renamed to ASP.NET Core 1.0.
* It consists of modular components (like separate code) with minimal overhead, so we retain flexibility while constructing our solutions.
* ASP.NET Core 6 Based on .NET Core 6 framework.
* From .NET 5.0, the word “Core” is dropped from its name.
* So, all frameworks from now on will be named as .NET 6, .NET 7 and so on.

Difference between .NET & .NET Core Frameworks:

What is .NET Framework?

* .Net is programming framework created by Microsoft that developers can use to create applications more easily. A framework is just a bunch of code that the programmer can call without having to write it explicitly.
* It is basically a collection of libraries.
* It has a number of pre-coded solutions that manage the execution of programs written specifically for the framework.

Difference:

.NET Framework .NET Core Framework

* Old framework New Framework
* Not Open-Source Open Source
* Only for windows platform Cross Platform
* Dependency Injection does not support DI support.
* .NET has no support for microservices .NET Core has support for microservices.
* Mobile Development does not support Mobile Development support.

.NET Framework Application models:

* The .NET Framework Application Model includes Win Forms, ASP.NET, and WPF, used for developing desktop and web apps.

.NET Core Framework Application model:

* .NET Core does not support desktop application development, focusing instead on the web, windows mobile and windows store with ASP.NET and Windows universal apps.

Prefer or choose .NET Core if:

* The project demands cross-platform integration.
* The project requires the development of microservices.
* The project relies heavily on CLI (Command Line Interface)

ASP.NET & ASP.NET Core:

* ASP.NET was the first version of the web-adapted .NET framework.
* ASP.NET Core is an improved version with richer functionality, a more comfortable interface, new libraries, and other distinctions.
* ASP.NET Core is the continuation of ASP.NET, an improved one.
* ASP.NET Core is available as an open source.

MVC With ASP.NET Core 6

Three basic components of MVC

* Model (Business Entities)
* View (Presentation Logic)
* Controller (Business Logic)
* 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.

Model

Updates Get Data

If No Data

View

Controller

Sees Request

USER

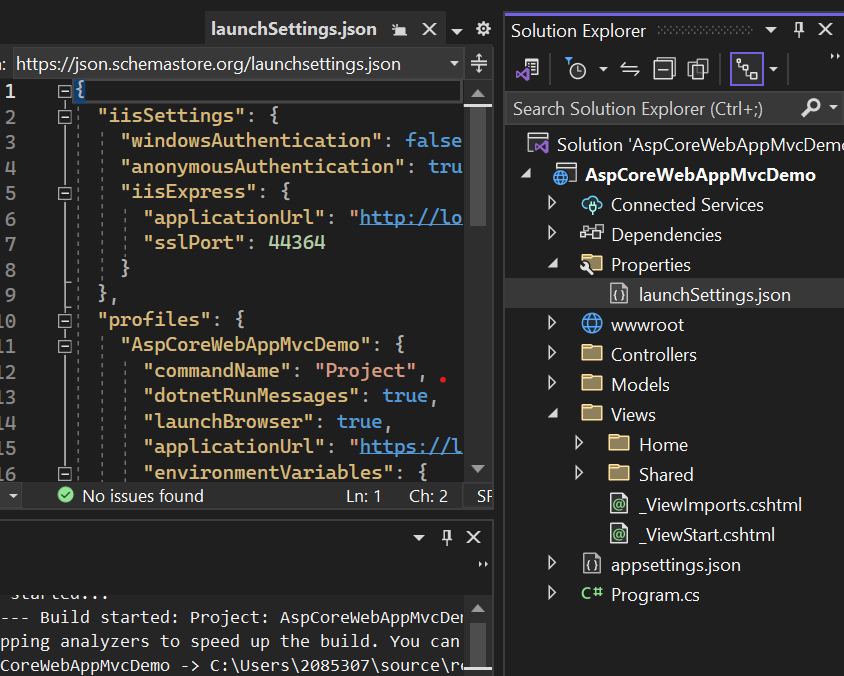
Creating first project in visual studio 2022:

* Open visual studio 2022 -> select new project -> choose ASP.NET Core web app MVC-> give name “AspCoreWebAppMVC” -> click on create.

**Project Structure Workflow:**

What is launchSettings.json in ASP.NET Core?

* In the properties folder in an asp.net core project, we can find the launchSettings.json file, which contains settings that control how our web app is stared on our development machine.
* The settings that are present within this file are going to be used when we run the .NET Core application either from Visual Studio or by using .NET core CLI.
* The most important point is that launchSettings.json file is only used within the local development machine.
* That means this file is not required when we are publishing our asp.net core application to the production server.



ASP.NET Core appsettings.json file:

* When we create an ASP.NET Core web application with an Empty project template or Razor Pages or MVC Template or Web API Template, then the visual studio automatically creates the appsettings.json file for us.
* The appsettings.json file is an application configuration file used to store configuration settings such as database connections strings, any application scope global variables, etc.

Middleware:

* A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application.
* Middleware in ASP.NET Core controls how our application responds to HTTP requests.
* Middleware are software components that are assembled into an application pipeline to handle requests and responses.
* It can also control how our application looks when there is an error.
* It is key piece in how we authenticate and authorize a user to perform specific actions.
* Each piece of middleware in asp.net core is an object, and each piece has a very specific, focused, and limited role.
* Ultimately, we need many pieces of middleware for an application to behave appropriately.
* Middleware has access to all the request and response.
* Order of middleware is very important.
* Middleware added first will process requests first and responses last.
* Middleware added later will process requests later and responses earlier.

Each middleware components can

* Process the incoming request.
* Call the next middleware in the pipeline.
* Process the outgoing response.

Run (): -

* Middleware defined using app.Run will never call subsequent middleware.
* It will take one parameter.
* It will not call next middleware. Run only one middleware at a time.
* Ex:

app.Run(async (context) =>

{

await context.Response.WriteAsync("Welecome to ASP.NET Core 6");

});

Use (): -

* The Use () method places a middleware in the pipeline and allows that middleware to pass control to the next item in the pipeline.
* It will call next middleware.
* It takes two parameters. One is context and other is next.
* Ex:

app.Use(async (context, next) =>

{

await context.Response.WriteAsync("Welcome to cognizant.");

await next(context);

});

app.Use(async (context,next) =>

{

await context.Response.WriteAsync("Welcome to asp.net core 6");

await next(context);

});

**Custom Middleware:**

* Here I am going to create a custom middleware for handling exception globally.
* First, we have to create a web api project. Inside this create a model folder and create a Category.cs class.
* Category.cs

namespace CustomMiddlewareExcaptionHandlling.Model

{

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

}

}

* Now create an empty API controller. Name as CategoriesController.cs.

using CustomMiddlewareExcaptionHandlling.Model;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace CustomMiddlewareExcaptionHandlling.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class CategoriesController : ControllerBase

{

[HttpGet]

public IActionResult GetCategories()

{

throw new NullReferenceException();

return Ok();

}

[HttpGet("{id}")]

public IActionResult GetCategoriesById(int id)

{

List<Category> list = new List<Category>();

list.Add(new Category { Id = 1,Name = "Abc" });

list.Add(new Category { Id = 2, Name = "Abc2" });

var cat = list.Where(x=>x.Id==id).First();

//if (id==0)

//{

// return BadRequest();

//}

return Ok(cat);

}

}

}

* In above controller file there an exception so now we are going to create a custom middleware which will handle that exception globally. For that purpose, we have to create a folder name as Middleware. In this folder create a class name as ExceptionMiddleware.cs. Before this we have to create a Model Class which is for error name as Error.cs.
* Error.cs

namespace CustomMiddlewareExcaptionHandlling.Model

{

public class Error

{

public string Title { get; set; }

public string Status { get; set; }

}

}

* ExceptionMiddleware.cs in this class we use RequestDelegate for calling the next middleware and create a InvokeAsync method for handling the exception.

using CustomMiddlewareExcaptionHandlling.Model;

using Newtonsoft.Json;

namespace CustomMiddlewareExcaptionHandlling.Middleware

{

public class ExceptionMiddleware

{

// Request Delegate will call the next middleware

private readonly RequestDelegate \_nextMidlleware;

public ExceptionMiddleware(RequestDelegate nextMidlleware)

{

\_nextMidlleware = nextMidlleware;

}

public async Task InvokeAsync(HttpContext context)

{

try

{

await \_nextMidlleware(context);

}

catch

{

context.Response.ContentType = "application/problem+json";

context.Response.StatusCode = StatusCodes.Status500InternalServerError;

var error = new Error

{

Title = "Error during running the code",

Status = "internal Server Error"

};

var result = JsonConvert.SerializeObject(error);

await context.Response.WriteAsync(result);

}

}

}

}

* Now we have to use this middleware in Program.cs class.
* Use app.UseMiddleware<ExceptionMiddleware>();

using CustomMiddlewareExcaptionHandlling.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseMiddleware<ExceptionMiddleware>();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

Routing in ASP.NET Core 6:

* The Routing in ASP.NET Core MVC application is a mechanism in which it will inspect (Jachna, Nigrani Karna) the incoming requests (i.e. URLs) and then mapped that request to the controllers and their action methods.
* ROUTING = URL + HTTP METHODS (GET, POST, PUT, DELETE)
* Localhost:PortName/Home/httpmethods
* This mapping is done by the routing rules which are defined for the application.
* We can do this by adding the Routing middleware to the request processing pipeline.
* So, the asp.net core framework maps or connects the incoming requests i.e. URLs to the Controllers action methods based on the routes configured in our application.

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

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

1. Convention Based Routing
2. Attribute-Based Routing

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 (i.e. URLs) to controllers and their action methods.
* In Asp.net Core MVC application, the Convention based Routes are defined within the Program.cs file.
* In Asp.net Core MVC application, it is the controller action method that is going to handle the incoming requests i.e. URLs.

--------------------------------------------------------------------------------------------------------------------------

Routing without MVC:

Steps:

1.Select ASP.NET Core Empty-> Click Next -> Select .net 6 -> click Create.

2.Add controller -> Right click on application -> click on Add -> add folder “Controllers” ->

3.Add Home controller (right click on Controllers folder – click on Add Choose controller – select MVC Controller – Empty)

4. Add view-> for adding view for IActionResult – Right click on IActionResult Index() method -> select Razor View click on Add – give name for view and click on Add -> It will create view for Home controller -> intex.cshtml inside views folder -> inside h1 tag give msg “Index from Home controller”

Next step inside Program.cs file:>

Comment this line: app.MapGet(“/”,()=>”Hello World” this line

Before “var app = builder.Build();” add below line:

Builder.Services.AddControllersWithViews();// this line is for view the controllers

Add middleware: app.MapDefaultControllerRoute();->It will call by default Home controller

If we don’t want to call home controller wanted to call another controller for this purpose, we need to add middleware:

app.MapControllerRoute(

name:”default”,

pattern:” {controller=Home}/{action=Index}/{id?}”);

In this convention Home Controller and Index action call by default and Id is optional

Now we want to add another controller name is: UserController-> create view for IActionResult Index ()-: view name is Index: Give msg: Index from User controller.

Now In Program.cs file:

App.MapControllerRoute(

Name:”default”,]

Pattern:” {controller=User}/{action=Index}/{id?}”);

It will call the User Controller and its index view.

Now we want to change action method for home controller so for that purpose we need to add.

Add method About:

Public IActionResult About () {

Return view ();}

Now we need to add view for this about method: Right click on About method and add view:- It will create view for About Action Method

App.MapControllerRoute(

Name:”default”,

Pattern:” {controller=Home}/{action=About}/{id?}”

);

Add other method Details (int id) in home controller:

Public int Details (int id)

{‘

return id;} ---------------------------------------------------------------------------------------------------------

Attribute Based Routing:

* In Attribute Based Routing**, [Route]** attribute is used to define the routes.
* For use attribute based Routing we need to add **builder.Services.AddControllersWithView();** in Program.cs file.
* Also Add **app.MapControllers();** in program.cs file.
* In Attribute-based routing, the route is determined 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.
* Changing the controller or action name does not require the route template to be changed.
* Token for controller is [controller].
* Token for action method is [action].
* We can use both tokens together[controller]/[action]
* **[Route("[controller]/[action]")]**

Map, MapGet, MapPost, MapPut, MapDelete

EndPoint:

* The MapGet method is used to define an endpoint.
* An endpoint is something that can be: Selected by matching the URL and HTTP method.
* Executed by running the delegate.
* app.UseRouting();
* app.Map(“/Home”,()=>”Hello World!”);
* app.MapGet(“/Home”,()=>”Hello World!-Get”);
* app.MapPost(“/Home”,()=>”Hello World! - Post”);
* app.MapPut(“/Home”,()=>”Hello World! -Put”);
* app.MapDelete(“/Home”,()=>”Hello World!-Delete”);
* Apart from these we can use.
* App.UseEndpoints(endpoints => {

endpoints.MapGet(“/Home”, async (context) =>{

Await context.Response.WriteAsync(“This is Home Page..GET”);

});

endpoints.MapPost(“/Home”, async (context) =>{

Await context.Response.WriteAsync(“This is Home Page..POST”);

});

endpoints.MapPut(“/Home”, async (context) =>{

Await context.Response.WriteAsync(“This is Home Page..PUT”);

});

endpoints.MapDelete(“/Home”, async (context) =>{

Await context.Response.WriteAsync(“This is Home Page..DELETE”);

});

});

ASP.NET Core App

**RoutingàControlleràAction MethodàView**

Browser

**URL**

Controller

Index About Details

Localhost:portNumber/Home localhost:PN/About localhost:PN/Details

In

These are called Action Methods. (Index, About, Details)

**Controller**

* Controller manages the flow of the application.
* A controller is used to define and group a set of actions.
* Is responsible for intercepting incoming requests and executing the appropriate application code.
* Controller is backbone of MVC.
* Communicates with the models of the application and selects the required view to be rendered for the request.
* Allows separating the business logic of the application from the presentation logic.
* Incoming requests are mapped to actions through routing.
* Controllers are basically C# classes that inherit from Microsoft.AspNetCore.Mvc.Controller.
* Controllers help in managing complete flow of application including accepting input and rendering appropriate output.
* Controllers along with its action method accepts incoming browser requests, retrieves required model information, and provides suitable responses.
* It is recommended that class name of a controller ends with suffix “Controller”.
* Are located in the root-level “Controllers” folder.

In an ASP.NET Core MVC application, a controller is responsible to:

* Locate the appropriate method to call for an incoming request.
* Validate the data of the incoming request before invoking the requested method.
* Retrieve the request data and passing it to requested method as arguments.
* Handle any exceptions that the requested method throws.
* Help in rendering the view based on the result of the requested method.

Action Methods:

* A controller class can contain one or more action methods, also known as controller actions.
* Actions are the methods in controller class which are responsible for returning the view or Json data.
* Action methods are public methods in Controller classes.
* Action methods are responsible for processing the requests that are sent to the controller.
* Action methods can return multiple type of data.
* By Default, it generates a response in the form of IActionResult interface or ActionResult Abstract Class.
* All the public methods inside a controller which respond to the URL are known as Action Methods.
* In ASP.NET MVC applications, we can create multiple action methods in a controller.
* We can invoke an action method by specifying a URL in web browser containing the name of the controller and the action method to invoke.

Rules that need to consider while creating an action method are as follows:

* They must be declared as public.
* Action must be public. It cannot be private or protected.
* They cannot be declared as static.

**IActionResult:**

* The IActionResult return type is an appropriate when multiple ActionResult return types are possible in an action.
* IActionResult and ActionResult works as a container for other action results.
* In that IActionResult is an interface and ActionResult is an abstract class that other action results inherit from.

Action Methods Returns:

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 resp.

ViewResult Represents an ActionResult that renders a view to the response.

ViewComponentResult An IActionResult which renders a view component to the response.

View – Razor Syntax & Razor View Engine:

**A View**

* A View provides the User Interface (UI) of the application to the user.
* A View is used to display content of an application and also to accept user inputs.
* View uses model data to create this UI.
* View contains model data to create this UI.
* View contains both HTML markup and C# code that runs on the Web server.
* View has a file extension .**cshtml.**

Razor

ViewàHTML Page Browser

HTML

<h2>Hello Keshav</h2>

<p>Good morning</p>

var name=”Programmer Analyst”;(C# Code)

Razor View

Engine

.cshtmlàCSharp +HTML

**Razor:**

* Razor is a syntax, based on the ASP.NET Core framework that allows creating views.
* Razor is used to simplify the process of creating views.
* Razor is simple and easy to understand for users who are familiar with the C# .NET programming languages.

Razor Engine:

* The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.
* Razor engine is used as the default view engine by the MVC Framework.
* Compiles a view of our application when the view is requested for the first time.
* Delivers the compiled view for subsequent requests until we make changes to the view.
* Does not introduce a new set of programming language but provides template markup syntax to segregate HTML markup and programming code in a view.
* First requires identifying the server-side code (C# code) from the markup code to interpret the server-side code embedded inside a view file.
* Uses the @ symbol, to separate the server-side code from the markup code.

While creating a Razor view, we should consider following rules:

* Start inline expression with @.
* Enclose code blocks between @{and}.
* Variables are declared with the var keyword.
* Enclose string in quotation marks.
* End a Razor code statement with semicolon (;).
* Use the. cshtml extension to store a razor view file that uses C# .NET as the programing language.
* Inline expression Statement ex:
* h1>Index</h1>
* <h1>Current Date and Time is @DateTime.Now</h1>
* <h1>Current Year is @DateTime.Now.Year</h1>
* Multi statement and Var keyword ex:

@{

var name = "Keshav";

var age = 26;

}

<h2>Name=@name</h2>

<h2>Age=@age</h2>

**Layout View - \_Layout. Cshtml (Master Page)**

* Layout View provides consistent looks for all the views in a web application.
* Layout View is as same as ASP.NET Web forms Master page.
* Extension of Layout View is. cshtml.
* Default name of Layout View is \_Layout.cshtml.
* Layout View files is usually located in View/Shared.
* An application can have multiple layout Views.
* Layout property is used to connect layout View with a view.

------------------------------------------------------------------------------------------------------------

Steps to create Layout:

Create Shared folder in View folderàRight click on shared folder àclick on AddàAdd new ItemàSelect Razor layout (name is: \_Layout.cshtml)àclick on create.

It will create a \_Layout.cshtml file. It has html code.

We can write code in body tag.

body>

<header style="background-color:yellow;">

<h1>Header Portion</h1>

</header>

@RenderBody()

<footer style="background-color:aqua;">

<h1>Footer Portion</h1>

</footer>

</body>

For use this \_Layout.cshtml file we need to add this in View files like Index.cshtml

@{

ViewData["Title"] = "Index";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

------------------------------------------------------------------------------------------------------------------

View Start File: \_ViewStart.cshtml

Why do we need \_ViewStart.cshtml file in ASP.NET Core MVC Application?

* As of now, we have used the Layout property to associate a view with a layout view as shown below.
* Layout=” ~/Views/Shared/\_Layout.cshtml”;
* Suppose we have 100 views in our application and all the 100 views want to use the same layout file.
* Then we need to set the Layout property in all the 100 views.
* This violates the DRY (Don’t Repeat Yourself) principle and has the following disadvantages.
* Redundant code
* Maintenance Overhead
* Commonly \_ViewStart.cshtml file is located in Views Folder.

**Steps to create \_ViewStart.cshtml file:**

* Right click on Views folderàClick on addà Add new ItemàSelect Razor View Start
* It will create \_ViewStart.cshtml file.

**Tasks:**

1. If we have Multiple Layout Views, then how we can use multiple layout views with View Start File?

* If we have Multiple Layout Views like \_Layout.cshtml and \_Layout.cshtml1 and we want to use \_Layout.cshtml in About.cshtml, Contact.cshtml (these are the views in views folder (inside Home folder) and \_Layout1.cshtml in index.cshtml
* Then we need to write Layout=”~/Views/Shared/\_Layout1.cshtml” in index.cshtml. And for other two view About.cshtml, Contact.cshtml we will create \_ViewStart.cshtml and write Layout = "\_Layout";.now, for Index.cshtml file It will give priority to \_Layout2.cshtml file not give priority to \_ViewStart.cshtml file layout.

Passing Data from Controller to View:

* In an ASP.NET Core MVC application, a controller typically performs the business logic of the application and needs to return the result to the user through a view.
* We can use the fo]llowing objects to pass data between controller and view:
* ViewData
* ViewBag
* TempData
* Strongly Typed Views

**ViewData:**

* Syntax:-> **ViewData[“<Key>”]=<Value>;**
* Key: Is a string value to identify the object present in ViewData.
* Value: Is the object present in ViewData. This object may be a string, object, list, array or a different type, such as int, char, float, double dateTime etc.
* It passes data from a controller to a view.
* Is a dictionary of objects that is derived from the ViewDataDictionary class.
* Some of the characteristics of ViewData are as follows:
* The life of a ViewData object exists only during the current request.
* The value of ViewData becomes null if the request is redirected.
* ViewData requires typecasting when we use complex data type to avoid error.
* Note: ViewData does not provide compile time error checking. For example – if we miss /spell keys, we wouldn’t know about the error only at runtime.

Example:

In Controller: HomeController.cs file we can write viewData Like

public IActionResult Index()

{

ViewData["data1"] = "CTS";

ViewData["data2"] = 26;

ViewData["data3"] = DateTime.Now.ToLongDateString();

string[] arr = { "Keshav", "Kumar", "Choudhary" };

ViewData["data4"] = arr;

ViewData["data5"] = new List<string> ()

{

"Cricket","Football","Hockey"

};

return View ();

}

Now for access these ViewData from controller We need to write code in Index.cshtml file like

<h1>Index Page</h1>

<p>This is my home page. </p>

@ViewData["data1"]

<br/>

@ViewData["data2"]

<br/>

@ViewData["data3"]

<br/>

@{

foreach (var item in (string[])ViewData["data4"])// typecasting

{

<h2>@item</h2>

}

}

@{

foreach (var item in (List<string>)ViewData["data5"])//typecasting

{

<h3>@item</h3>

}

}

**ViewBag:**

* ViewBag is also used to transfer data from a controller to a view.
* Syntax: **ViewBag.<PropertyName>=<Value>;**
* Property: Is a string value that represents a property of ViewBag.
* Value: Is the value of the property of ViewBag, Value may be a string, object, list, array or a different type, such as int, char, float, double DateTime etc.
* ViewBag is a dynamic type of property of the base class of all the controllers, which is the Controllerbase class.
* ViewBag is a dynamic data type, which internally uses ViewData to store values.
* ViewBag exists only for the current request and becomes null if the request is redirected.
* ViewBag is a dynamic property based on the dynamic features introduces in c#4.0.
* ViewBag does not require typecasting when we use complex data type. Like array, collections, List etc.
* Note: ViewBag does not provide compile time error checking. For example- if we mis-spell keys we wouldn’t get any compile time errors. We get to know about the error only at runtime.

Example:

In HomeController.cs file: we can write code for ViewBag like

public class HomeController : Controller

{

public IActionResult Index()

{

ViewBag.data1 = "CTS";

ViewBag.data2 = 26;

ViewBag.data3 = DateTime.Now.ToShortDateString();

string[] arr = { "Keshav", "Kumar", "Chy" };

ViewBag.data4 = arr;

ViewBag.data5 = new List<string>()

{

"Cricket","Football","Hockey"

};

ViewData["myName"] = "Keshav Kumar";

ViewBag.data6 = "Welcome to my cts account.";

}

}

Now for access ViewBag code we need to write code in Index.cshtml file

@{

ViewData["Title"] = "Index";

// Layout = "~/Views/Shared/\_Layout.cshtml";

Layout = "~/Views/Shared/\_Layout2.cshtml";

}

<h1>Index Page</h1>

<p>This is my home page. </p>

@ViewBag.data1

<br/>

@ViewBag.data2

<br/>

@ViewBag.data3

<br/>

@{

foreach (var item in ViewBag.data4)

{

<h2>@item</h2>

}

}

@{

foreach (var item in ViewBag.data5)

{

<h3>@item</h3>

}

}

@ViewBag.myName

<br/>

@ViewData["data6"]

**TempData:**

* It passes data from a controller to a view.
* Syntax: TempData[<Key>]=<Value>;
* Key: Is a String value to identify the object present in TempData.
* Value: Is the object present in TempData.
* TempData is used only for current or subsequent requests as it is very short-lived instance.
* Redirectly is the only case when users can rely on TempData.
* When redirecting, current request is killed, and a new request is created on the server the redirected view.
* Sharing data between the controller actions are done through the asp.net mvc TempData dictionary.
* TempData is a Dictionary object derived from the TempDataDictionary class.
* TempData stores data as key-value pairs.
* TempData value must be type cast before use.
* Check for null values to avoid runtime error.
* TempData allows passing data from the current request to the subsequent request during request redirection.
* TempData in ASP.NET MVC can be used to store temporary data which can be used in the subsequent request.
* TempData will be cleared out after the completion of a subsequent request.
* Call TempData.Keep() to keep all the values of TempData in a third request.

Example:

In HomeController.cs file :

public class HomeController : Controller

{

public IActionResult Index()

{

ViewData["data1"] = "View Data";

ViewBag.data2 = "View Bag";

TempData["data3"] = "Temp Data";

TempData.Keep();

TempData["data4"] = new List<string>()

{

"Cricket","Football","Hockey"

};

// TempData["data4"] = null;

return View();

}

public IActionResult About()

{

TempData.Keep();

return View();

}

public IActionResult Contact()

{

return View();

}

In the Index.cshtml file:

<h1>Index Page</h1>

<p>This is my home page. </p>

@ViewData["data1"]

@ViewBag.data2

@TempData["data3"]

@{

if (TempData["data4"] != null)

{

foreach (var item in (List<string>)TempData["data4"])

{

<h3>@item</h3>

}

}

else

{ <h2>No data found...</h2>}}

In About.cshtml file:

<h1>About Page</h1>

<p>This is my About Us Page. </p>

@ViewData["data1"]

@ViewBag.data2

@TempData["data3"]

In Contact.cshtml file:

<h1>Contact Page</h1>

<p>This is my Contact US page.</p>

@ViewData["data1"]

@ViewBag.data2

@TempData["data3"]

**Models in ASP.NET Core 6:**

* A model is a class with .cs (for C#) as an extension having both properties and methods.
* The Models in ASP.NET Core MVC contains set of classes that are used to represent the domain data (we can say the business data) as well as it also contains logic to manage the domain/business data.
* So, in simple terms, we can say that the models in ASP.NET Core MVC Application are used to manage the data.
* STUDENT

rollNo

Name

Gender

Standard

* Models are used to set or get the data.
* If our application does not have data, then there is no need for a model.
* If our application has data, then we need a model.
* If we are working with any Web Application that is based on MVC Design Pattern, then in that MVC Application, three things are common i.e. Model, View, and Controller.
* The Controllers are used to manage the overall flow of the application. Models are responsible for the data and these data are used on Views.
* Views are basically the HTML pages that get rendered into the browser of the client.
* We can perform operations like insert, update, delete by using Models.
* Note: It is not mandatory, but it is good programming practice to store all the model classes within the Models folder.

Example for creating model class and how it uses in controller and its views:

Steps:

1. First, we need to create Model class name is “StudentModel.cs”:

namespace ModelsInASPCore.Models

{ public class StudentModel{

public int rollNo { get; set; }

public string Name { get; set; }

public string Gender { get; set; }

public int Standard { get; set; }

}}

1. Now we need to use this StudentModel class in HomeController.cs file:

public IActionResult Index()

{

var students = new List<StudentModel>

{

new StudentModel{ rollNo= 1,Name = "Keshav", Gender = "Male",Standard = 11},

new StudentModel{ rollNo= 2,Name = "Madhav", Gender = "Male",Standard = 12},

new StudentModel{ rollNo= 3,Name = "Pranav", Gender = "Male",Standard = 10}

};

ViewData["myStudents"] = students;

return View();

}

1. Now we have to create view for this Index () method:

@{

ViewData["Title"] = "Home Page";

var students = ViewData["myStudents"] as List<StudentModel>;

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

<table>

<tr>

<th>ROLL NO</th>

<th>NAME</th>

<th>GENDER</th>

<th>CLASS</th>

</tr>

@foreach (var std in students)

{

<tr>

<td>@std.rollNo</td>

<td>@std.Name</td>

<td>@std.Gender</td>

<td>@std.Standard</td>

</tr>

}

</table>

Note for Table Tag:

**<table> tag** – A table is created by using <table> tag. In the <table> element the table is built row by row. A row is present in a <tr> tag, which stands for ‘table row’. And each cell is then written inside the row element using a <td> tag, which stands for ‘table data’. Each cell must have a <td> or a <th> tag to display correctly.

**Repository Pattern:**

Repository Pattern in ASP.NET Core 6

Repository

Data Access Layer

Controller

Class Interface Model

Implement Abstract Methods

USERàIIS SERVERà CONTROLLERàß REPOSITORYàßDBCONTEXT (Data access layer)àDATABASE

(Repository interacts with data access layer using LINQ)

* With the Repository pattern, we create an abstraction layer between the data access and the business logic layer of an application.
* By using it, we are promoting a more loosely coupled approach to access our data from the database.
* Also, the code is cleaner and easier to maintain and reuse.

Repository creation in Visual Studio:

Steps:

1. So, we have a StudentModel.cs class where we already created property for students with property of rollNo, Name, Gender, Standard.
2. Now we need to create a Repository folder where we will create IStudent.cs Interface and Student.cs class file.
3. IStudent.cs:

using ModelsInASPCore.Models;

namespace ModelsInASPCore.Repository

{

public interface IStudent

{

List<StudentModel> getAllStudents();

StudentModel getStudentById(int id);

}}

1. This interface contains abstract method, so we have to create a class for given body.

For that purpose, we must create a class name as StudentRepository.cs class.

using ModelsInASPCore.Models;

namespace ModelsInASPCore.Repository

{public class StudentRepository: IStudent{

private List<StudentModel> DataSource()

{

return new List<StudentModel>

{

new StudentModel { rollNo = 1, Name = "Keshav", Gender = "Male", Standard = 11 },

new StudentModel { rollNo = 2, Name = "Madhav", Gender = "Male", Standard = 12 },

new StudentModel { rollNo = 3, Name = "Pranav", Gender = "Male", Standard = 10 },

new StudentModel { rollNo = 4, Name = "Suraj", Gender = "Male", Standard = 7 },

new StudentModel { rollNo = 5, Name = "Pranay", Gender = "Male", Standard = 1 },

};}

public List<StudentModel> getAllStudents()

{

return DataSource();

}

public StudentModel getStudentById(int id)

{

return DataSource().Where(x=>x.rollNo==id).FirstOrDefault();

}}}

1. Before moving to next step we have to comment all the code from Index () method except return View(). Now next step is in HomeController.cs file. In this we have to create 2 methods according to IStudent.cs file one is getAllStudents () and getById (). For accessing StudentRepository create a variable private readonly StudentRepository \_studentRepository = null; now this variable we must need to initialize through the constructor:

using Microsoft.AspNetCore.Mvc;

using ModelsInASPCore.Models;

using ModelsInASPCore.Repository;

using System.Diagnostics;

namespace ModelsInASPCore.Controllers

{

public class HomeController: Controller

{

private readonly ILogger<HomeController> \_logger;

private readonly StudentRepository \_studentRepository = null;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger.

\_studentRepository = new StudentRepository();

}

public List<StudentModel> GetAllStudents()

{

return \_studentRepository.getAllStudents();

}

public StudentModel getById(int id)

{

return \_studentRepository.getStudentById(id);

}

* In the Index.cshtml comment the <table> </table> codes.

**How to Pass Model Data from Controller to View using Strongly Typed Views?**

**Strongly Typed Views:**

* Strongly Typed views or strongly Typed Object is used to pass data from controller to a view.
* The View which binds with any model is called as strongly typed view.
* We can bind any class as a model to view.
* We can access model properties on that view.
* We can use data associated with model to render controls.
* The view that is designed by targeting specific model class object then that view is called “Strongly Typed View”.
* In Strongly typed view, view is bind with corresponding model class object or list of objects.

Steps:

Create a project select ASP.NET Core web app(MVC) name as StronglyTypedViewASPCore

1. First step is: we have to create a model class name as “Employee.cs” class where we write property for Employee:

namespace StronglyTypedViewASPCore.Models

{public class Employee

{

public int EmpId {get; set;}

public string EmpName {get; set;}

public string Designation {get; set;}

public int Salary {get; set;}

}}

1. Go to Controller folder in this we need write code in IActionResult Index () method in HomeController.cs. We can pass single data or List of data.

public IActionResult Index ()

{

//Employee obj = new Employee()

//{

// EmpId = 101,

// EmpName="Keshav",

// Designation="PA",

// Salary=33000

//};

var myEmployee = new List<Employee>

{

new Employee { EmpId=1,EmpName="Keshav",Designation="PA",Salary=33000},

new Employee { EmpId=2,EmpName="Madhav",Designation="SM",Salary=450000}

};

return View(myEmployee);}

1. Now we must go to the View of Index () method. For bind to Strongly type view we need to add Employee model class in Index.cshtml file.

@\* @model Employee \*@@\*@model StronglyTypedViewASPCore.Models.Employee\*@

@model List<Employee>

@\*@model IEnumerable<StronglyTypedViewASPCore.Models.Employee> \*@

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

@{

foreach (var item in Model)//all the property all save in Model so we use Model

{

<h2>@item.EmpId</h2>

<h2>@item.EmpName</h2>

<h2>@item.Designation</h2>

<h2>@item.Salary</h2>

<p>---------------------</p>

}}

<@\* h2>@Model.EmpId</h2>

<h2>@Model.EmpName</h2>

<h2>@Model.Designation</h2>

<h2>@Model.Salary</h2> \*@

**\_ViewImports.cshtml in ASP.NET Core 6:**

* In ASP.NET Core MVC Application, the \_Viewimports.cshtml file provides a mechanism to include the directives globally for Razor Pages/Views so that we don’t have to add them individually in each and every page.
* \_ViewImports.cshtml is a feature in ASP.NET Core MVC that enables you to add common namespaces, directives, and other elements to multiple views within your application without having to add these namespaces, directives, and other elements to every individual view file. It essentially serves as a way to include common code that should be available to all views in the same directory or in directories.
* ViewImports.cshtml file supports the following directives:
* @addTagHelper
* @tagHelperPrefix
* @removeTagHelper
* @namespace
* @inject
* @model
* @using
* Steps to create \_ViewImports.cshtml file:

Create ASP.NET Core Empty projectàName – ASPCoreViewImportsàAdd Controllers folderàAdd Controller Name-“HomeController.cs”àIn this we have by default Index () method, create 2 more method About () and Contact () àAdd Views for each methodàIndex.cshtml, About.cshtml and Contact.cshtmlàCreate Models folder àAdd Student.cs classàGive properties for Student classàlike Id,Name,Genderà

Now in HomeController.cs file write code for model in each action method.

public IActionResult Index()

{

List<Student> students = new List<Student>

{

new Student{ Id=1,Name="Keshav",Gender="Male"},

new Student{ Id=2,Name="Anjali",Gender="Female"},

new Student{ Id=3,Name="Gulshan",Gender="Male"}

};

return View(students);

}

For using models in each action’s view like Index.cshtml About.cshtml,and

Contact.cshtml we have to create a \_ViewImport.cshtml file for writing common directivesàRight Click on Views folderàAdd new Itemà Search Razor view importsà

It will add \_ViewImports.cshtml fileà write code @using ASPCoreViewImports.Models

àNow go to views file Index.cshtml, About.cshtml and Contact.cshtml and write @model List<Student>

**Tag Helpers:**

* Tag helpers are basically special attributes provided by Asp.net Core.
* Tag Helpers enable (saksham) server-side components to participate in creating and rendering HTML elements in Views.
* Tag helpers are a new feature and similar ton HTML helpers (MVC 5), which help us render HTML.
* There are many built-in Tag Helpers for common tasks, such as creating forms, hyperlinks, loading assets etc.
* Tag Helpers are authored in C#, and they target HTML elements based on the element name.
* For example, the built-in LabelTagHelper can target the HTML <label> element when the LabelTagHelper attribute are applied.
* Before start working with tag helpers, make sure it included namespace for tag helpers in our ViewImports file.
* Microsoft.AspNetCore.Mvc.TageHelpers
* Add this line in our \_ViewImports file.
* **@addTagHelper\*, Microsoft.AspNetCore.Mvc.TagHelpers**
* Add this line in our \_ViewImports file.
* @addTagHelper\*, Microsoft.AspNetCore.Mvc.TagHelpers
* @addTagHelper is a directive.
* **Example for creating hyperlinks using TagHelper:**

Steps:

Create a Asp.net Core Mvc project. In this we have a HomeController.cs file.

Which have Index () method and it returns Index.cshtml view. Now we create one more method name as Edit (int id) it returns id and also create a View for this method. Now write code for hyperlinks in Index.cshtml file for rendering Edit page.

@{

ViewData["Title"] = "Home Page";}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

@\* <a href="/Home/Contact">Contact Page 1 </a>

<br />

@Html.ActionLink("Contact Page 2","Contact","Home")

<br />

<a href="@Url.Action("Contact","Home")">Contact Page 3</a>

<br />

<a asp-controller="Home" asp-action="Contact">Contact Page 4</a>

\*@

<a href="/Home/Edit/1">Edit Page 1 </a>

<br />

@Html.ActionLink("Edit Page 2","Edit","Home",new {id=1})

<br />

<a href="@Url.Action("Edit","Home",new {id=1})">Edit Page 3</a>

<br />

<**a** **asp-controller**="Home" **asp-action**="Edit" **asp-route-id**="1">Edit Page 4</**a**>

* <a href="/Home/Edit/1">Edit Page 1 </a> à This is normal html code for creating hyperlink
* @Html.ActionLink("Edit Page 2","Edit","Home",new {id=1})à This is used in MVC 5 for creating hyperlink
* <a href="@Url.Action("Edit","Home",new {id=1})">Edit Page 3</a> à This is also used in MVC 5 for creating hyperlinks
* <**a** **asp-controller**="Home" **asp-action**="Edit" **asp-route-id**="1">Edit Page 4</**a**>à

This is TagHelper in MVC 6 which creates hyperlinks.

* Above all are working same. All are using for creating hyperlinks.

**Image Tag Helper:**

Browser cache memory

Browser Server

View

(Image)

Understanding the Browser Cache

* In general, when we visit a web page and if that web page contains some image, most of the modern web browser cache the images for later use.
* In the future, when we revisit to that web page, then the browser loads the images from cache instead of downloading the images from the server.
* In most of the caches, this is not an issue as the images are not changes that often for a web page.
* How to Disable Browser cache?

Go to … icon in browserà go to more toolsàDeveloper toolsà Networkà Check Disable Cache

* In order to use the Image Tag Helper in ASP.NET Core Application, we need to add the

asp-append-version attribute to the <img> tag and need to set the value to true.

* Steps:
* Add image folder in wwwroot folder. copy images to this folder. In Index.cshtml file write the image tag helper code.
* <img src="~/Images/Cognizant.jpg" asp-append-version="true" height="200" width="400" alt="New Image"/>
* Image tag helper provides cache busting behavior within asp.net core application.

**Creating Form Using Tag Helper:**

* The asp-for attribute is arguably one of the most common tag helpers that we will encounter, and its primary purpose is to handle binding a specific property to the element that it decorates.
* asp-for tag helper is used for an <input>, it sets the name attribute, so that it can be bound to model class.
* E.g. asp-for = ”Movie.Year” when retrieving data from a POST request.
* asp-for sets the id, name and validation related attributes.
* Steps for creating form using Tag helper:
* Create a Model Employee.cs class which have 4 properties Name, Age, Designation, Salary

public class Employee

{

public string Name {get; set;}

public Gender Gender {get; set;}

public int Age {get; set; }

public string Designation {get; set;}

public int Salary {get; set;}

public string Married {get; set;}

public string Description {get; set;}

}

Public enum Gender

{Male, Female}

* Write form code in index.cshtml file

<div class="container">

<div class="row">

<div class="col-sm-4">

<**form** class="d-grid" **asp-action**="Index" **asp-controller**="Home" method="post">

<**input** **asp-for**="Name" placeholder="Enter Name" class="form-control" />

<br />

<**input** **asp-for**="Age" placeholder="Enter Age" class="form-control" />

<br />

<**input** **asp-for**="Designation" placeholder="Enter Designation" class="form-control" />

<br />

<**input** **asp-for**="Salary" placeholder="Enter Salary" class="form-control" />

<br />

<input type="submit" value="Submit" class="btn btn-outline-primary btn-block"

</**form**>

</div>

</div></div>

* **Tag Helpers are:**
* The Form Tag Helper.
* The Form Action Tag helper.
* The Input Tag Helper.
* The Text area Tag Helper.
* The Label tag Helper.
* The Validation Tag Helpers.
* The Select Tag Helper.

**Creating DropDownList, Radio Buttons, Label, TextArea Using Tag Helpers**

**DropDownList:**

* In Model Employee.cs class we must make an enum for Gender: public enum Gender {Male, Female}
* In the Index.cshtml file we must write code for drop down list for gender

<**select** **asp-for**="Gender" class="form-control" **asp-items**="Html.GetEnumSelectList<Gender> ()">

<**option** **value**="">Select Gender</**option**>

</**select**>

**Radio Buttons:**

* In Model class we have a property Married so, we make a radio button YES or No
* In Index.cshtml file we must write code for radio button

<label>

Married:

</label>

<div class="form-check">

<**input** class="form-check-input" **type**="radio" **asp-for**="Married" **value**="Yes">

<label class="form-check-label" for="flexRadioDefault1">

Yes

</label>

</div>

<div class="form-check">

<**input** class="form-check-input" **type**="radio" **asp-for**="Married" **value**="No">

<label class="form-check-label" for="flexRadioDefault2">

No

</label> </div>

**Label:**

* <**label** **asp-for**="Name">Enter Name:</**label**>

**TextArea:**

* <**textarea** **asp-for**="Description" class="form-control" placeholder="EnterDescription" rows="5"> </**textarea**>

**Model Binding:**

* Model Binding in ASP.NET Core MVC maps data from HTTP requests to action method parameters.
* The parameters may be simple types such as strings, integers, or floats, or they may be complex types like Student, Employee, Customer, Order etc.

**How model binding works**

* When MVC receives an HTTP request, it routes it to a specific action method of a controller.
* It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action methods parameters.
* In addition to route values MVC will bind data from various parts of the request, and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:
* **Form values:** These are from values that go in the HTTP request using the POST method.
* **Route Values**: The set of route values provided by routing.
* **Query Strings:** The query string part of the URL.
* Note: form values, route data, and query strings are all stored as name-value pairs (Key-Value pairs).

Example:

* Create ASP.NET Core Web App (MVC) in the HomeController.cs file create a method Details () which takes two parameters id and name and this method return the id and name.
* public string Details (int id, string name)

{

return "Id is: " + id + " Name is: " + name;

}

* When we run this application, we have to pass url like this
* <https://localhost:7037/Home/Details/1?name=Keshav> // (This is Query Strings)

**Model Validation:**

* Often the data entered by the user is not valid and cannot be saved into the database.
* The entered data may contain a type or user may intentionally enter the inappropriate data.
* Hence, we need to validate the user input before storing it in the database.
* The ASP.NET Core gives us ModelValidator, which uses the validation attributes to validate the model, which makes our task easier.
* We also take at ModelState and how to use it. Finally, we look at the list of Validation attributes.
* The Form Data is posted to Controller action is automatically mapped to the action parameter by the Model Binder.
* When the HTTP request arrives Model Binder is invoked before passing the control to controller action method.
* The Model Binder not only binds the value to the action parameter but also validates them by using the Model Validator.
* The ModelValidator runs after the model binding and runs a series of validations on each property of the model based on the attributes that set on the model property.
* These attributes are called Validation attributes and contain the code, which is used by the ModelValidator to validate the model.
* The Model needs to be validated for the correctness.
* These validations can be done at the client side before sending data to the server or at the server side when the data is received from the client.

**Explicitly Validating a Model**

* Once we received the model in the controller, we can validate the model programmatically.
* If (string.IsNullOrEmpty(model.Name)){
* //Validation failed
* //Send the list of errors to client
* }

**Built-in attributes**

* [Required]
* [StringLength]
* [EmailAddress]
* [Range]
* [RegularExpression]
* [Compare]
* [Phone]
* [Url]

All these validation attributes are located in **System.ComponentModel.DataAnnotations;**

**[Required]:**

* Validates that the field is not null.
* [Required(ErrorMessage ="Name is Must")]

**[StringLength]:**

* Validates that a string property value doesn’t exceed a specified length limit**.**
* [StringLength(15,MinimumLength =3, ErrorMessage ="Name must be within 3 to 15 characters")].

**[EmailAddress]:**

* Validates that the property has an email format.

**[Range]:**

* Validates that the property value falls within a specified range.

**[RegularExpression]:**

* Validates that the property value matches a specified regular expression.

**[Compare]:**

* Validates that the properties in model match.

**[Url]:**

* Validates that the property has a URL format.

**[MinLength]:**

* Validates that the property minimum length.

**[MaxLength]:**

* Validates that the property maximum length.

Validation Tag Helpers:

* There are 2 validation tag helpers in Asp.Net Core MVC.
* Validation Message Tag Helper: It displays the validation message for a single property of a Model. E.g.: **asp-validation-for**
* Validation Summary Tag Helper: It displays a summary of validation messages for all the properties of a Model. E.g.: **asp-validation-summary**
* <div asp-validation-summary="All"></div>

Example of Model Validation:

Steps:

* Create a ASP.NET Core web app (MVC) project. In this project we have Model, Controller and View folder by default. In the Model folder create a Student.cs class. Which have all the property and all validation for Student. In the HomeController.cs file we have IActionResult Index () method but create one more method IActionResult Index (Student std). It takes one parameter Student std and put [HTTPPost] route on upper portion of this method. In the Index.cshtml file we write code forViews.
* Student.cs file code:

using System.ComponentModel.DataAnnotations;

namespace ValiadationAttributesASPCore.Models

{

public class Student

{

[Required(ErrorMessage ="Name is Must")]

// [StringLength(15,MinimumLength =3, ErrorMessage ="Name must be within 3 to 15 characters")]

// [MaxLength(15)]

[MinLength(3,ErrorMessage ="Minimum length must be 3 chars")]

public string Name { get; set; }

[Required(ErrorMessage ="Email Is Must")]

// [EmailAddress]

[RegularExpression("[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?")]

public string Email { get; set; }

[Required(ErrorMessage = "Age is Must")]

[Range(10,50,ErrorMessage ="Age Must be within 10 to 50")]

public int? Age { get; set; }

//[Required(ErrorMessage = "Password is Must")]

//[RegularExpression("(?!^[0-9]\*$)(?!^[a-zA-Z]\*$)^([a-zA-Z0-9]{8,15})$")]

//public string Password { get; set; }

//[Required(ErrorMessage ="Confirm Password is Must")]

//[Compare("Password",ErrorMessage ="Both Passwords Must be Identical")]

//[Display(Name="Enter Confirm Password")]

//public string ConfirmPassword { get; set; }

[Required(ErrorMessage ="Number is Must")]

[RegularExpression("^\\(?([0-9]{3})\\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$",ErrorMessage ="Invalid Mobile Number")]

public String INumber { get; set; }

[Required(ErrorMessage =" Website URL is Must")]

[Url(ErrorMessage ="Invalid URL")]

public string WebsiteURL { get; set; }

}

}

* HomeController.cs file code:

using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;

using ValiadationAttributesASPCore.Models;

namespace ValiadationAttributesASPCore.Controllers

{

public class HomeController : Controller

{

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

public IActionResult Index()

{

return View();

}

[HttpPost]

public IActionResult Index(Student std)

{

if (ModelState.IsValid) // This if block will clear data after //submit data on browser

{

ModelState.Clear();

}

return View();

//if (ModelState.IsValid)

//{

// return "Name is:" + std.Name;

//}

//else

//{

// return "Validation Failed..";

//}

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

* Index.cshtml file code:

@model ValiadationAttributesASPCore.Models.Student;

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

<div class="container">

<div class="row">

<div class="col-sm-4">

<**div** **asp-validation-summary**="All" style="color:red;">

</**div**>

<**form** **asp-controller**="Home" **asp-action**="Index" method="post" >

<**input** **asp-for**="Name" class="form-control" placeholder="Enter Name"/>

@\*< span asp-validation-for="Name" style="color:red;"></span> \*@

<br />

<**input** **asp-for**="Email" class="form-control" placeholder="Enter Email" />

@\* <span asp-validation-for="Email" style="color:red;"></span> \*@

<br />

<**input** **asp-for**="Age" class="form-control" placeholder="Enter Age" />

@\* <span asp-validation-for="Age" style="color:red;"></span> \*@

<br />

@\* <input asp-for="Password" class="form-control" placeholder="Enter password" />

<span asp-validation-for="Password" style="color:red;"></span>

<br />

<label asp-for="ConfirmPassword"></label>

<input asp-for="ConfirmPassword" class="form-control" placeholder="Enter confirm Password" />

<span asp-validation-for="ConfirmPassword" style="color:red;"></span> \*@

<br />

<**input** **asp-for**="INumber" class="form-control" placeholder="Enter Number" />

@\* <span asp-validation-for="INumber" style="color:red;"></span> \*@

<br />

<**input** **asp-for**="WebsiteURL" class="form-control" placeholder="Enter Website URL" />

@\* <span asp-validation-for="WebsiteURL" style="color:red;"></span> \*@

<br />

<input type="submit" value="Submit" class="btn btn-outline-primary"/>

</**form**>

</div>

</div>

</div>

Introduction to Entity Framework Core in ASP.NET Core

* To address the data access requirements of ASP.NET CORE MVC application, we can use an ORM (OBJECT RELATIONAL MAPPING) framework.

An ORM framework

* Simplifies the process of accessing data from applications.
* ORM is a tool for storing data from domain objects to relational database.
* Domain Objects means model classes and relational database means SQL Server.

**Entity Framework Core:**

* Entity Framework Core is the new version of Entity Framework after EF 6, but it is redesigned.
* It is open source, lightweight, extensible and a cross-platform version of Entity Framework.
* Entity Framework Core is a data access technology.
* Entity Framework is an Object/Relational Mapping (ORM) framework.
* It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.
* EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5 + framework-based applications.

**Entity Framework Core Development Approaches**

* EF Core supports two development approaches.
* 1.) Code-First
* 2.) Database-First

**Code First Approach of EF Core:**

* In the code-first approach the Entity Framework core creates database objects based on model classes that we create to represent application data.
* It is the most common approach implemented in ASP.NET CORE MVC Framework.
* Allows us to develop our application by coding model classes and properties and delegate the process of creating the database objects to the Entity Framework Core.
* The code-first approach allows us to define our own model by creating custom classes.
* Then, we can create database based on the models.

Model Class

Migration

Database Tables

Migration\_name

Student

Run

DbContext Class

StudentDbContext

**Steps**

1st Steps:

* Install 3 packages in our ASP.NET Core MVC application.
* 1. Microsoft.EntityFrameworkCore.SqlServer
* 2. Microsoft.EntityFrameworkCore.Tools
* 3. Microsoft.EntityFrameworkCore.Design

**Microsoft.EntityFrameworkCore.Tools:**

* They are primarily used to manage Migrations and to scaffold a DBContext.

**Microsoft.EntityFrameworkCore.Design:**

* It contains all the design-time logic for Entity framework core.

2nd Step:

* Create a Model Class.
* Create a DbContext Class.

**DbContext**

* The DbContext class is an integral part of Entity Framework.
* This is the class that we use in our application code to interact with the underlying database.
* It is this class that manages the database connection and is used to retrieve and save data in the database.
* An instance of DbContext represents a session with the database which can be used to query and save instances of our entities to a database.
* DbContext is a combination of the unit of work and repository patterns.
* DbContext can be used to define the database context class after creating a model class.
* DbContext co-ordinates with Entity Framework and allows us to query and save the data in the database.
* Uses the DbSet <T> type to define one or more properties where, t represents the type of an object that needs to be stored in the database.

**DbContextOptions in EF Core:**

* For the DbContext class to be able to do any useful work, it needs an instance of the DbContextOptions class.
* The DbContextOptions instance carries configuration information such as the connection string, database provider to use etc.

3rd Step:

* Create a Connection String in appsettings.json file.

4th Step:

* Registering Connection String in Program.cs File.

5th Step:

* Add a migration and run the migration.

**Code First approach steps and codes in Visual Studio:**

* Create a ASP.NET Core Web App (MVC) project name as “CodeFirstASPCore6”.
* Install 3 packages in our CodeFirstASPCore6 application.

1. Microsoft.EntityFrameworkCore.SqlServer

2. Microsoft.EntityFrameworkCore.Tools

3. Microsoft.EntityFrameworkCore.Design

* In the Models folder add class name as “Student.cs”. And write property for Student class. And also write column name, TypeName.

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace CodeFirstASPCore6.Models

{

public class Student

{

[Key]

public int Id { get; set; }

[Column("StudentName",TypeName ="varchar(100)")]

public string Name { get; set; }

[Column("StudentGender", TypeName ="varchar(20)")]

public string Gender { get; set; }

public int Age { get; set; }

}

}

* For this model Student.cs class we have to add another class which is DbContext name is
* StudentDbContext.cs class. It is this class that manages the database connection and is used to retrieve and save data in the database.
* Uses the DbSet <T> type to define one or more properties where, it represents the type of an object that needs to be stored in the database.

using Microsoft.EntityFrameworkCore;

namespace CodeFirstASPCore6.Models

{

public class StudentDbContext : DbContext

{

public StudentDbContext(DbContextOptions options):base(options)

{

}

public DbSet<Student> Students { get; set; }

}

}

* Write ConnectionStrings in the appsettings.json file. In below format.

**"ConnectionStrings": {**

**"dataBaseConnectionString": "Server=ServerNameOfSSMS;Database=DatabaseName;Trusted\_Connection=True;"**

**}**

Trusted\_connection is for windows authentication.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"ConnectionStrings": {

"dataBaseConnectionString": "Server=LTIN231224\\SQLSERVER;Database=CodeFirstDB;Trusted\_Connection=True;"

},

"AllowedHosts": "\*"

}

* Next step is Registering Connection String in Program.cs File.
* Write below code before var app = builder.Build ().

**var provider = builder.Services.BuildServiceProvider();**

**var config = provider.GetRequiredService<IConfiguration>();**

**builder.Services.AddDbContext<StudentDbContext>(item => item.UseSqlServer(config.GetConnectionString("dataBaseConnectionString")));**

using CodeFirstASPCore6.Models;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

var provider = builder.Services.BuildServiceProvider();

var config = provider.GetRequiredService<IConfiguration>();

builder.Services.AddDbContext<StudentDbContext>(item => item.UseSqlServer(config.GetConnectionString("dataBaseConnectionString")));

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

* Now Add a migration and run the migration. Go to the Tools à Nuget Package Managerà Package Manager Console.
* In Package Manager Console write below line and run this command: add-migration migrationName

**add-migration CodeFirstCreateDB**

**update-database**

* After running this command, it will create a migration folder which have all the codes for creating database related things.
* Now Go to SQL Server Management Studio and check database. There is dbo.Students.
* If we want to add one more column in Students table then again, we have to go to the Model class Student.cs file and add the property for that column for example I want to add Standard column, so for that purpose I need to write property for Standard in Student.cs file.

public int Standard { get; set; }

* Now again we have to perform migration command.

add-migration CodeFirstAddClass

update-database

* Now from SSMS we will insert some data in this table. Right click on dbo.Students and select Edit top 200 rows. And insert some data in these columns and execute.
* Now come in Controllers folder in HomeController.cs file we have to write some code.

Create a parametrized constructor of HomeController class and pass

(StudentDbContext studentDB) parameter to it. Add below line in HomeController.cs file.

private readonly StudentDbContext studentDB;

public HomeController(StudentDbContext studentDB)

{

this.studentDB = studentDB;

}

* using CodeFirstASPCore6.Models;

using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;

namespace CodeFirstASPCore6.Controllers

{

public class HomeController : Controller

{

//private readonly ILogger<HomeController> \_logger;

//public HomeController(ILogger<HomeController> logger)

//{

// \_logger = logger;

//}

private readonly StudentDbContext studentDB;

public HomeController(StudentDbContext studentDB)

{

this.studentDB = studentDB;

}

public IActionResult Index()

{

var stdData=studentDB.Students.ToList();

return View(stdData);

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });}}}

* Now next step is in Views folder. First, we need to delete by default Index.cshtml file from Views folder.

Now we need to create a View for Index () method.

* Right click on index () method à Add Viewà Select Razor Viewà click on Addà

Fill the information for razor Viewà

View name - Index

Template – List

Model class – Student (CodeFirstASPCore6.Models)

DbContext class – StudentDbContext (CodeFirstASPCore6.Models)

Click on Add

* It will create a view for index method Now we can Run this project.

Creating CRUD Application using Code-First Approach:

* We are going to creating CRUD Application in above project “CodeFirstASPCore6”.
* Just edit existing Models, Controllers, View for Get, Post, Put, Delete method. Now We are writing code for all those folder.

**Models Folder: Student.cs class**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace CodeFirstASPCore6.Models

{

public class Student

{

[Key]

public int Id { get; set; }

[Column("StudentName",TypeName ="varchar(100)")]

[Required]

public string Name { get; set; }

[Column("StudentGender", TypeName ="varchar(20)")]

[Required]

public string Gender { get; set; }

[Required]

public int Age { get; set; }

[Required]

public int Standard { get; set; }

}

}

**StudentDbContext.cs file:**

using Microsoft.EntityFrameworkCore;

namespace CodeFirstASPCore6.Models

{

public class StudentDbContext : DbContext

{

public StudentDbContext(DbContextOptions options):base(options)

{

}

public DbSet<Student> Students { get; set; }

}

}

**Controllers folder: HomeController.cs file**

using CodeFirstASPCore6.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Diagnostics;

namespace CodeFirstASPCore6.Controllers

{

public class HomeController : Controller

{

//private readonly ILogger<HomeController> \_logger;

//public HomeController(ILogger<HomeController> logger)

//{

// \_logger = logger;

//}

private readonly StudentDbContext studentDB;

public HomeController(StudentDbContext studentDB)

{

this.studentDB = studentDB;

}

public async Task<IActionResult> Index()

{

var stdData= await studentDB.Students.ToListAsync();

return View(stdData);

}

public IActionResult Create()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(Student std)

{

if (ModelState.IsValid)

{

await studentDB.Students.AddAsync(std);

await studentDB.SaveChangesAsync();

TempData["insert\_success"] = "Inserted..";

return RedirectToAction("Index","Home");

}

return View(std);

}

public async Task<IActionResult> Details(int? id)

{

if (id == null || studentDB.Students == null )

{

return NotFound();

}

var stdData = await studentDB.Students.FirstOrDefaultAsync(x=>x.Id==id);

if(stdData == null)

{

return NotFound();

}

return View(stdData);

}

public async Task<IActionResult> Edit(int? id)

{

if (id == null || studentDB.Students == null)

{

return NotFound();

}

var stdData = await studentDB.Students.FindAsync(id);

if (stdData == null)

{

return NotFound();

}

return View(stdData);

}

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int? id, Student std)

{

if(id!= std.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

studentDB.Update(std);

await studentDB.SaveChangesAsync();

TempData["update\_success"] = "Updated..";

return RedirectToAction("Index", "Home");

}

return View(std);

}

public async Task<IActionResult> Delete(int? id)

{

if (id == null || studentDB.Students == null)

{

return NotFound();

}

var stdData = await studentDB.Students.FirstOrDefaultAsync(x => x.Id == id);

if (stdData == null)

{

return NotFound();

}

return View(stdData);

}

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int? id)

{

var stdData = await studentDB.Students.FindAsync(id);

if(stdData != null)

{

studentDB.Students.Remove(stdData);

}

TempData["delete\_success"] = "Deleted..";

await studentDB.SaveChangesAsync();

return RedirectToAction("Index", "Home");

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

**Views folder:**

* Create views for all the Controllers method like Index, Create, Details, Edit, Delete.
* All these views create automatically. We have to choose like below.
* In HomeControllers.cs file à Right click on Create (Student std) method à Add Viewà Select Razor Viewà click on Addà

Fill the information for razor Viewà

**View name - Create**

**Template – Create**

**Model class – Student (CodeFirstASPCore6.Models)**

**DbContext class – StudentDbContext (CodeFirstASPCore6.Models)**

**Click on Add**

* Same steps done for all the Controllers method like Edit, Details, Delete.

**Database First Approach of Entity framework Core in ASP.NET Core 6:**

**Existing Database Command Generated Model Classes**

**& Tables Scaffold DbContext + DbContext Class**

* **I**n the database-first approach the Entity Framework core creates model classes and properties corresponding to the existing database objects, such as table and columns.
* The database-first approach is applicable in scenario where a database already exists for the application.
* Steps:
* **1st Step:**

Install 3 Packages in our ASP.NET Core MVC application.

1. Microsoft.EntityFrameworkCore.SQLServer
2. Microsoft.EntityFrameworkCore.Tools
3. Microsoft.EntityFrameworkCore.Design

**2nd Step:**

* Execute a command for Scaffold DbContext.
* **Scaffold-DbContext “server=ServerName;database=DatabaseName;trusted\_connection=true”**

**Microsoft.EntityframeworkCore.SqlServer**

**-OutputDir Models**

* Above command will generate model class and DbContext class automatically.
* If we update our database, then how we can update our Model and DbContext.
* By using this command:
* **Scaffold-DbContext “server=ServerName;database=DatabaseName;trusted\_connection=true”**

**Microsoft.EntityframeworkCore.SqlServer**

**-OutputDir Models -force**

**3rd Step:**

* Move Connection string from DbContext class to appsettingd.json file.

**4th Step:**

* Registering Connection String in Program.cs.
* Now in Database first approach we are going to implement CRUD operations using automatically create all action method and views.
* For that purpose, we have to delete HomeControllers.cs file and Home Folder from Views Folder.
* Steps to create automatically code:
* Right Click on Controllers folderàAdd Controllerà Choose MVC Controller with Views, using EntityFrameworkà Click on ADD à It will generate a Template where we have to select Model Class, DbContext class and Controller name.

**Model Class Students (DatabaseCodeFirstEFCore.Models)**

**DbContext class CodeFirstDBContext (DatabaseCodeFirstEFCore.Models)**

**Controller name HomeController**

* So, it will create all action method and Views.

**Session in ASP.NET Core**

* What is Session?

Sessions are one of the several ways to manage state in an ASP.NET Core application.

A Session state is a mechanism for the storage of user data across the application. It creates and stores an identifier in a cookie, which is then used to retrieve the session data on the server.

* It is a state management technique.
* Session state is an ASP.NET Core scenario for storage of user data while the user browsers 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.
* Critical application data should be stored in the user database and cached in session only as a performance optimization.
* The session is specific to the browser, Sessions aren’t shared across browsers.
* Sessions are deleted when the browser session ends.
* Sessions are Server-Side.
* Session is also used to pass data within the ASP.NET Core MVC application and unlike TempData.
* It persists for its expiration time (default time is 20 minutes but it can be increased or decreased).
* Session is valid for all requests, not for a single redirect.
* A session state stores application-specific data in key-value pairs.
* A session state stores user-specific information for an ASP.NET MVC application.
* However, the scope of session state store is limited to the current browser session.
* When many users access an application simultaneously, then, each of these users will have a different session state.

Steps to Implement Session:

**Step: 1**

* Add below line before build,

**“builder.Services.AddSession();”**

* Add below line after build,

**“app.UseSession();”**

**Step:2**

* Create Session Variable. Before creating variable add **using Microsoft.AspNetCore.Http**.
* **HttpContext.Session.SetString(“MyKey”,”Keshav”);**
* We can set different type of values in a session.

**Step:3**

* Access Session Variable.
* **HttpContext.Session.GetString(“MyKey”);**

Step:4

* If you want to access session directly in view, not in action method then you have to use **HttpContextAccessor.**

Creating Login with Database, Session & Logout In ASP.NET Core 6

Phases:

1. Setup Session in ASP.NET Core App.
2. Implementing Database First Approach.
3. Login & Logout Task.

Setup Session:

* Add below line before build in program.cs,

“builder.Services.AddSession();

* Add below line after build in program.cs,

“app.UseSession();”

Implementing DB First:

1st Step:

* Install 3 package in our ASP.NET Core MVC application.
* Microsoftg.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tool
* Microsoft.EntityFrameworkCore.Design

2nd Step:

* Execute a command for Scafofold DBContext.
* Scaffold-DbContext “server=ServerName;database=DatabaseName;trusted\_connect=true”
* Above command will generate model class and DBContext class automatically.

3rd Step:

* Move Connection String from DbContext class to appsettings.json file.

4th Step:

* Registering Connection String in Program.cs file.
* var provider = builder.Services.BuildServiceProvider();
* var config = provider.GetRequiredService<IConfiguration>();
* builder.Services.AddDbContext<MyDBContext>(item => item.UseSqlServer(config.GetConnectionString("databaseConnectionString")));

Login & Logout

* Create action methods for login and logout.
* Create session variable in login action method and how we can access it.

**Codes for Login Project:**

User.cs:

* using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace LoginFormASPCore6.Models

{

public partial class User

{

public int Id { get; set; }

[Required]

public string Name { get; set; } = null!;

[Required]

public string Gender { get; set; } = null!;

[Required]

public int? Age { get; set; }

[Required]

public string Email { get; set; } = null!;

[DataType(DataType.Password)]

[Required]

public string Password { get; set; } = null!;

}

}

* **HomeController.cs:**

using LoginFormASPCore6.Models;

using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;

using Microsoft.AspNetCore.Http;

namespace LoginFormASPCore6.Controllers

{

public class HomeController : Controller

{

private readonly MyDBContext context;

public HomeController(MyDBContext context)

{

this.context = context;

}

public IActionResult Index()

{

return View();

}

public IActionResult Login()

{

if (HttpContext.Session.GetString("UserSession") != null)

{

return RedirectToAction("Dashboard");

}

return View();

}

[HttpPost]

public IActionResult Login(User user )

{

var myUser = context.Users.Where(x => x.Email == user.Email && x.Password == user.Password).FirstOrDefault();

if(myUser != null)

{

HttpContext.Session.SetString("UserSession", myUser.Email);

return RedirectToAction("Dashboard");

}

else

{

ViewBag.Message = "Login Failed..";

}

return View();

}

public IActionResult Dashboard()

{

if(HttpContext.Session.GetString("UserSession") != null)

{

ViewBag.MySession = HttpContext.Session.GetString("UserSession").ToString();

}

else

{

return RedirectToAction("Login");

}

return View();

}

public IActionResult Logout()

{

if (HttpContext.Session.GetString("UserSession") != null)

{

HttpContext.Session.Remove("UserSession");

return RedirectToAction("Login");

}

return View();

}

public IActionResult Register()

{

return View();

}

[HttpPost]

public async Task<IActionResult> Register(User user)

{

if(ModelState.IsValid)

{

await context.Users.AddAsync(user);

await context.SaveChangesAsync();

TempData["Success"] = "Registered Successfully";

return RedirectToAction("Login");

}

return View();

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

Views File:

* **Index.cshtml:**

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

* **Login.cshtml:**

@model LoginFormASPCore6.Models.User

@{

ViewData["Title"] = "Login";

}

<h1>Login</h1>

<h4>User</h4>

<hr />

<div class="row">

<div class="col-md-4">

@{

if(ViewBag.Message != null)

{

<p style="color:red;">@ViewBag.Message</p>

}

if(TempData["Success"] != null)

{

<p style="color:green;">@TempData["Success"]</p>

}

}

<**form** **asp-action**="Login">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="Email" class="control-label"></**label**>

<**input** **asp-for**="Email" class="form-control" />

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

</div>

<div class="form-group">

<**label** **asp-for**="Password" class="control-label"></**label**>

<**input** **asp-for**="Password" class="form-control" />

<**span** **asp-validation-for**="Password" class="text-danger"></**span**>

</div>

<div class="form-group">

<input type="submit" value="Login" class="btn btn-primary" />

</div>

</**form**>

<br />

<p>

<**a** **asp-controller**="Home" **asp-action**="Register">Not Registered Yet ? Click Here for SignUp</**a**>

</p>

</div>

</div>

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

* **Dashboard.cs file:**

@{

ViewData["Title"] = "Dashboard";

}

<h1>Dashboard</h1>

<h2>@ViewBag.MySession</h2>

<**a** **asp-controller**="Home" **asp-action**="Logout">Logout</**a**>

* **Register.cs file:**

@model LoginFormASPCore6.Models.User

@{

ViewData["Title"] = "Register";

}

<h1>Register</h1>

<h4>User</h4>

<hr />

<div class="row">

<div class="col-md-4">

<**form** **asp-action**="Register">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="Name" class="control-label"></**label**>

<**input** **asp-for**="Name" class="form-control" />

<**span** **asp-validation-for**="Name" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="Gender" class="control-label"></**label**>

<**input** **asp-for**="Gender" class="form-control" />

<**span** **asp-validation-for**="Gender" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="Age" class="control-label"></**label**>

<**input** **asp-for**="Age" class="form-control" />

<**span** **asp-validation-for**="Age" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="Email" class="control-label"></**label**>

<**input** **asp-for**="Email" class="form-control" />

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

</div>

<div class="form-group">

<**label** **asp-for**="Password" class="control-label"></**label**>

<**input** **asp-for**="Password" class="form-control" />

<**span** **asp-validation-for**="Password" class="text-danger"></**span**>

</div>

<div class="form-group">

<input type="submit" value="Register" class="btn btn-primary" />

</div>

</**form**>

</div>

</div>

<div>

<**a** **asp-action**="Login">Login Form</**a**>

</div>

* **Program.cs file:**

using LoginFormASPCore6.Models;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

builder.Services.AddSession();

var provider = builder.Services.BuildServiceProvider();

var config = provider.GetRequiredService<IConfiguration>();

builder.Services.AddDbContext<MyDBContext>(item => item.UseSqlServer(config.GetConnectionString("databaseConnectionString")));

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseSession();

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

**Creating Drop Down List in ASP.NET Core 6:**

* Code for creating Drop Down List:
* We are going to use enum for Gender data. Here we are not using database.
* Here I am going to create a public enum Gender {Male, Female} in HomeController.cs file.
* In the Index method create a instance of Gender as a List<SelectListItem>.
* HomeController.cs file code:

using DropDownASPCore6.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using System.Diagnostics;

using System.Net.Sockets;

namespace DropDownASPCore6.Controllers

{

public enum Gender

{

Male,

Female

}

public class HomeController : Controller

{

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

public IActionResult Index()

{

List<SelectListItem> Gender = new()

{

new SelectListItem{Value="Male", Text="Male" },

new SelectListItem{Value="Female", Text="female", },

};

ViewBag.Gender = Gender;

return View();

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

* Create a view for Index method.
* Index.cshtml file code:

@using DropDownASPCore6.Controllers;

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

@\* <label class="form-label">Select Gender</label>

@Html.DropDownList("Gender",

Html.GetEnumSelectList<Gender>(),"Select Gender",

new {@class="form-control"})

\*@

@\*

<label class="form-label">Select Gender</label>

@Html.DropDownList("Gender",

ViewBag.Gender,"Select Gender",

new {@class="form-control"})

\*@

<label class="form-label">Select Gender:</label>

<**select** **name**="Gender" **asp-items**="@ViewBag.Gender" class="form-control">

<**option** **value**="">Select Gender</**option**>

* </**select**>

**How to Bind Multiple Models with a Single View in ASP.NET Core:**

* If we have multiple model class for example Student.cs and Teacher.cs class. Now we want to bind these classes for that purpose we need to create a third model class name as SchoolViewModel.cs class where we create property for Student and Teacher and use this in View Model class for binding multiple Models.
* Here I am writing code:
* **Student.cs file:**

namespace ViewModelASPCore6.Models

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public string Gender { get; set; }

public int Standard { get; set; }

}

}

* **Teacher.cs file:**

namespace ViewModelASPCore6.Models

{

public class Teacher

{

public int Id { get; set; }

public string Name { get; set; }

public string Qualification { get; set; }

public int Salary { get; set; }

}

}

* **SchoolViewModel.cs file:**

namespace ViewModelASPCore6.Models

{

public class SchoolViewModel

{

public List<Student> myStudents { get; set; }

public List<Teacher> myTeachers { get; set; }

}

}

* **HomeController.cs file:**

using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;

using ViewModelASPCore6.Models;

namespace ViewModelASPCore6.Controllers

{

public class HomeController : Controller

{

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger)

{

\_logger = logger;

}

public IActionResult Index()

{

List<Student> students = new List<Student>

{

new Student { Id = 1,Name="Keshav",Gender="Male",Standard=12},

new Student { Id = 2,Name="Madhav",Gender="Male",Standard=13},

new Student { Id = 1,Name="Mukund",Gender="Male",Standard=10},

};

List<Teacher> teachers = new List<Teacher>

{

new Teacher { Id =1,Name="Hans",Qualification="BTech",Salary=50000},

new Teacher { Id = 2,Name="Raj",Qualification="MCA",Salary=30000},

new Teacher { Id = 3,Name="Aman",Qualification="BA",Salary=10000},

};

SchoolViewModel svm = new SchoolViewModel()

{

myStudents=students,

myTeachers=teachers

};

return View(svm);

}

public IActionResult Privacy()

{

return View();

}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

}

}

* **Index.cshtml file for View:**

@model ViewModelASPCore6.Models.SchoolViewModel

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

<div class="container">

@if (Model.myStudents.Any())

{

<h2>STUDENTS DATA</h2>

<table class="table table-bordered table-dark table-hover table-striped">

<thead>

<tr>

<th>ID</th>

<th>NAME</th>

<th>GENDER</th>

<th>CLASS</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.myStudents)

{

<tr>

<td>@item.Id </td>

<td>@item.Name </td>

<td>@item.Gender </td>

<td>@item.Standard </td>

</tr>

}

</tbody>

</table>

}

else

{

<h2>No Students Found..</h2>

}

</div>

<br />

<div class="container">

@if (Model.myTeachers.Any())

{

<h2>TEACHERS DATA</h2>

<table class="table table-bordered table-dark table-hover table-striped">

<thead>

<tr>

<th>ID</th>

<th>NAME</th>

<th>QUALIFICATION</th>

<th>SALARY</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.myTeachers)

{

<tr>

<td>@item.Id </td>

<td>@item.Name </td>

<td>@item.Qualification </td>

<td>@item.Salary </td>

</tr>

}

</tbody>

</table>

}

else

{

<h2>No Teachers Found</h2>

}

</div>

**Partial Views in ASP.NET Core 6:**

* Partial View represents a sub-view/nested view/ inner view of a main view.
* Partial View allows you to reuse common markups across the different views of the application.
* We can use partial views in different views.
* Partial views cannot be used separately, we must attach partial in some other view.
* Partial view extension is. cshtml like a view.
* When we have to use some html markup on some pages not all pages then we can use partial view.
* Remember: A Partial View is a Part of a View File.
* We can use <partial name=”\_EmployeeNames”> TagHelper in View file for using partial view.
* We create partial view inside a Shared folder.
* Steps for creating partial view:

We are going to create a ASP.NET Core MVC project name as “PartialViewASPCore”. It will create a MVC project with Index.cshtml file and many more.

Inside Shared folder we create a Partial View name as \_EmployeeNames for that purpose right click on Shared filder àADDàViewàRazor Viewà

View name - \_Employee

Template- Empty

Options:

Mark check on Create as a partial view à click on ADD

It will create a Partial View

* \_EmployeeNames.cshtml file:

<h2>List of Employee Names</h2>

<ul>

<li>Keshav</li>

<li>Madhav</li>

<li>Pranav</li>

<li>Suraj</li>

</ul>

* Now use this partial; view in Index.cshtml file.

@{

ViewData["Title"] = "Home Page";

}

\<div class="text-center">

<h1 class="display-4">Welcome</h1>

<p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>

</div>

<**partial** **name**="\_EmployeeNames"/>

<**partial** **name**="\_EmployeeNames" />

<**partial** **name**="\_EmployeeNames" />

* Now we can use this partial view in other View file as well for Example About.cshtml file:

@{

ViewData["Title"] = "About";

}

<h1>About</h1>

<**partial** **name**="\_EmployeeNames" />

**Partial View with Models (Strongly Typed Partial View)**

**Strongly Typed Partial View:**

* It is a combination of 2 concepts in ASP.NET Core MVC.
* 1. Strongly Typed View: The View which binds with a model is called as strongly typed view.
* We can bind any class as a model to view.
* We can access model properties on that view.
* 2. Partial View
* The Partial View which binds with any model is called as strongly typed partial view.
* Strongly Typed Partial View is also known as dynamic partial views.
* There are 2 types of Partial Views.
* 1.Static Partial Views: Partial View with no Model (Static data).
* 2.Dynamic Partial Views: Partial View with Model (Dynamic data).

**Introduction of ASP.NET Core Web API (Restful APIs) in ASP.NET Core 6:**

**API:**

* API stands for Application Programming Interface.
* The HTTP Protocol is utilized by Web API, a concept that is used to increase an applications capability.
* A web-based application alone is no longer sufficient to reach all of its users or clients.
* WEB API is an application programming interface that is used to enable communication or interaction with software components with each other.
* Web API is the enhanced form of the web application to provide services on different devices like laptop, mobile, and others.
* A Web API is a set of endpoints (URL) that allow different software applications to communication with each other over the internet.

**ASP.NET Core Web API:**

* ASP.NET Core Web API is a framework developed by Microsoft that allows you to build and create RESTful APIs using the ASP.NET Core platform.
* It is designed to facilitate the development of lightweight, fast, and efficient APIs that can be consumed by various clients, such as web applications, mobile apps, or other services.
* Web API allows users to access a particular resource using HTTP protocol.
* Web API is often used to provide an interface for web sites and client applications to have data access.
* Web API can be used to access data from a database and save data back to the database.

**RESTFUL Architecture:**

* ASP.NET Core Web API follows the principles of Representational State Transfer (REST).
* It uses HTTP methods like GET, POST, PUT, and DELETE to interact with URLs (Uniform resource locators).
* It is introduced in 2000 by Roy Fielding.
* In REST architecture, a REST Server simply provides access to resources and the REST client accesses and presents the resources.
* Here each resource is identified by URLs/Global IDs.
* REST uses various representations to represent a resource like Text, JSON and XML.
* JSON is now the most popular format being used in Web Services.
* A RESTful system consists of:
* Client who requests for the resources.
* Server who has the resources.

**What is RESTful Services?**

* REST architecture pattern specifies a set of constraints that a system adhere to.

1. Uniform Interface
2. Client Server
3. Stateless
4. Cacheable
5. Layered system.
6. Code on demand

**Restful API**

RESOURCE VERB RESULT

/Students GET Gets list of students

/ Students/1 GET Gets student with id = 1

/Students POST Creates a new student

/Students/1 PUT Updates student with id = 1

/Students/1 DELETE Deletes student with id = 1

**What is Swagger?**

* Swagger is used to describe Restful APIs.
* Swagger allows developer to create interactive and human-readable API documentation.
* Swagger is used to Test APIs.

**What is ControllerBase?**

* The ControllerBase class is a base class for controller in ASP.NET Core that handles HTTP requests.
* It provides a set of common properties and methods controllers use to handle HTTP requests and generate HTTP responses.

**What is [ApiController] attribute?**

* The [ApiController] attribute enables a few features including attribute routing requirement, automatic model validation and binding source parameter inference.

**ASP.Net Core Web API CRUD Operation Using Entity Framework Core & SQL Server:**

**Objectives:**

**First Part:**

* Create ASP.Net Core Web API CRUD operations using Entity Framework Core & SQL Server. (API Server).

**Second Part:**

* Consume ASP.NET Core Web API for CRUD operations in ASP.NET Core MVC Web Application. (Client)

**StringContent class:**

* StringContent class creates a formatted text appropriate for the http server/client communication.
* After a client request, a server will respond with a HttpResponseMessage and that response will need a content, which can be created with the StringContent class.

**What is Authentication & Authorization?**

* **Authentication:** It is process of checking the identity of the user that accessing our application.
* **Authorization:** It is process of validating privileges to access a resource of the application.
* After successful login to the application (authentication), authorization mechanisms checks whether login user has privileges to access the application resource.

**First Part:**

* Create ASP.Net Core Web API CRUD operations using Entity Framework Core & QL Server. (API Server).
* Here I am going to use database first approach for perform CRUD operation.
* Create a ASP.NET Core Web API project name as “ASPCore6WebAPICRUD”.

**1st Step:**

* Install 3 package in our ASP.NET Core Web API.
* Microsoftg.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tool
* Microsoft.EntityFrameworkCore.Design

**2nd Step:**

* Execute a command for Scafofold DBContext.
* Scaffold-DbContext “server=ServerName;database=DatabaseName;trusted\_connect=true”
* **Scaffold-DbContext "server=LTIN231224\SQLSERVER; database=MyDB; trusted\_connection=true;" Microsoft.EntityFrameworkCore.SQLServer -OutputDir Models**
* Above command will generate model class name as User.cs and Student.cs and DBContext class name as MyDBContext.cs class automatically.
* Here we are using Student.cs class.
* **Student.cs class:**

using System;

using System.Collections.Generic;

namespace ASPCore6WebAPICRUD.Models

{

public partial class Student

{

public int Id { get; set; }

public string StudentName { get; set; } = null!;

public string StudentGender { get; set; } = null!;

public int? Age { get; set; }

public string Standard { get; set; } = null!;

public string FatherName { get; set; } = null!;

}

}

**3rd Step:**

* Move Connection String from DbContext class to appsettings.json file.

**"ConnectionStrings": {**

**"databaseConnectionString": "server=LTIN231224\\SQLSERVER; database=MyDB; trusted\_connection=true;"**

**},**

**4th Step:**

* Registering Connection String in Program.cs file.
* var provider = builder.Services.BuildServiceProvider();
* var config = provider.GetRequiredService<IConfiguration>();
* builder.Services.AddDbContext<MyDBContext>(item => item.UseSqlServer(config.GetConnectionString("databaseConnectionString")));
* **Program.cs file:**

using ASPCore6WebAPICRUD.Models;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var provider = builder.Services.BuildServiceProvider();

var config = provider.GetRequiredService<IConfiguration>();

builder.Services.AddDbContext<MyDBContext>(item => item.UseSqlServer(config.GetConnectionString("databaseConnectionString")));

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

**5th Step:**

* Delete by default file from controller.
* Create Controller name as StudentAPIController.cs. here we are going to write all HTTP methods for performing CRUD operation.
* **StudentAPIController.cs :**

using ASPCore6WebAPICRUD.Models;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

namespace ASPCore6WebAPICRUD.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class StudentAPIController : ControllerBase

{

private readonly MyDBContext context;

public StudentAPIController(MyDBContext context)

{

this.context = context;

}

[HttpGet]

public async Task<ActionResult<List<Student>>> GetStudents()

{

var data= await context.Students.ToListAsync();

return Ok(data);

}

[HttpGet("{id}")]

public async Task<ActionResult<Student>> GetStudentById(int id)

{

var student=await context.Students.FindAsync(id);

if (student == null)

{

return NotFound();

}

return student;

}

[HttpPost]

public async Task<ActionResult<Student>> CreateStudent(Student std)

{

await context.Students.AddAsync(std);

await context.SaveChangesAsync();

return Ok(std);

}

[HttpPut("{id}")]

public async Task<ActionResult<Student>> UpdateStudent(int id, Student std)

{

if(id !=std.Id)

{

return BadRequest();

}

context.Entry(std).State = EntityState.Modified;

await context.SaveChangesAsync();

return Ok(std);

}

[HttpDelete("{id}")]

public async Task<ActionResult<Student>> DeleteStudent(int id)

{

var std = await context.Students.FindAsync(id);

if(std == null)

{

return NotFound();

}

context.Students.Remove(std);

await context.SaveChangesAsync();

return Ok();

}

}

}

* Run this project.

**Second Part:**

* Consume ASP.NET Core Web API for CRUD operations in ASP.NET Core MVC Web Application. (Client)
* Create a ASP.NET Core Web App (MVC) project name as CRUDAppUsingASPCoreWebAPI6.
* Create a Model class Student.cs. for property of Student class, we are doing some steps.
* Run the previous web api project “ASPCore6WebAPICRUD” in the background. On the browser we fetch the data through GET by id. It will give the response like below.

**{**

**"id": 1,**

**"studentName": "Keshav Kumar",**

**"studentGender": "Male",**

**"age": 26,**

**"standard": "BTech",**

**"fatherName": "Sunil Kumar Choudhary"**

**}**

* Copy these codes from the swagger.
* Now In CRUDAppUsingASPCoreWebAPI6 project go to the Edit option click on Paste Special option choose Paste as JSON Classes. It will create a Student Property in Student.cs class.
* **Student.cs class:**

using System.ComponentModel.DataAnnotations;

namespace CRUDAppUsingASPCoreWebAPI6.Models

{

public class Student

{

public int id { get; set; }

[Required]

public string studentName { get; set; }

[Required]

public string studentGender { get; set; }

[Required]

public int age { get; set; }

[Required]

public string standard { get; set; }

[Required]

public string fatherName { get; set; }

}

}

* Create a Controller name as StudentController.cs file.
* In StudentController file use url variable which hold the url of the ASPCore6WebAPICRUD project.
* **private string url = "https://localhost:7276/api/StudentAPI/";**
* **StudentController.cs file:**

using CRUDAppUsingASPCoreWebAPI6.Models;

using Microsoft.AspNetCore.Mvc;

using Newtonsoft.Json;

using System.Text;

namespace CRUDAppUsingASPCoreWebAPI6.Controllers

{

public class StudentController : Controller

{

private string url = "https://localhost:7276/api/StudentAPI/";

private HttpClient client= new HttpClient();

[HttpGet]

public IActionResult Index()

{

List<Student> students = new List<Student>();

HttpResponseMessage response = client.GetAsync(url).Result;

if (response.IsSuccessStatusCode)

{

string result = response.Content.ReadAsStringAsync().Result;

var data = JsonConvert.DeserializeObject<List<Student>>(result);

if (data!=null)

{

students = data;

}

}

return View(students);

}

[HttpGet]

public IActionResult Create()

{

return View();

}

[HttpPost]

public IActionResult Create(Student std)

{

string data=JsonConvert.SerializeObject(std);

StringContent content= new StringContent(data,Encoding.UTF8,"application/json");

HttpResponseMessage response = client.PostAsync(url, content).Result;

if(response.IsSuccessStatusCode)

{

TempData["insert\_message"] = "Student Added..";

return RedirectToAction("Index");

}

return View();

}

[HttpGet]

public IActionResult Edit(int id)

{

Student std = new Student();

HttpResponseMessage response=client.GetAsync(url + id).Result;

if (response.IsSuccessStatusCode)

{

string result = response.Content.ReadAsStringAsync().Result;

var data= JsonConvert.DeserializeObject<Student>(result);

if(data!=null)

{

std = data;

}

}

return View(std);

}

[HttpPost]

public IActionResult Edit(Student std)

{

string data = JsonConvert.SerializeObject(std);

StringContent content = new StringContent(data, Encoding.UTF8, "application/json");

HttpResponseMessage response = client.PutAsync(url + std.id, content).Result;

if (response.IsSuccessStatusCode)

{

TempData["update\_message"] = "Student Updated..";

return RedirectToAction("Index");

}

return View();

}

[HttpGet]

public IActionResult Details(int id)

{

Student std = new Student();

HttpResponseMessage response = client.GetAsync(url + id).Result;

if (response.IsSuccessStatusCode)

{

string result = response.Content.ReadAsStringAsync().Result;

var data = JsonConvert.DeserializeObject<Student>(result);

if (data != null)

{

std = data;

}

}

return View(std);

}

[HttpGet]

public IActionResult Delete(int id)

{

Student std = new Student();

HttpResponseMessage response = client.GetAsync(url + id).Result;

if (response.IsSuccessStatusCode)

{

string result = response.Content.ReadAsStringAsync().Result;

var data = JsonConvert.DeserializeObject<Student>(result);

if (data != null)

{

std = data;

}

}

return View(std);

}

[HttpPost,ActionName("Delete")]

public IActionResult DeleteConfirmed(int id)

{

HttpResponseMessage response = client.DeleteAsync(url + id).Result;

if (response.IsSuccessStatusCode)

{

TempData["delete\_message"] = "Student Deleted..";

return RedirectToAction("Index");

}

return View();

}

}

}

* Create view for all the action method in the ViewàStudent folder like Create.cshtml, Index.cshtml, Edit.cshtml, Details.cshtml, Delete.cshtml.
* **Index.cshtml file:**

@model IEnumerable<CRUDAppUsingASPCoreWebAPI6.Models.Student>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<**a** **asp-action**="Create">Create New</**a**>

</p>

@{

if (TempData["insert\_message"]!=null)

{

<div class="alert alert-success alert-dismissible fade show" role="alert">

<strong>Success</strong> @TempData["insert\_message"]

<button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>

</div>

}

if (TempData["update\_message"] != null)

{

<div class="alert alert-success alert-dismissible fade show" role="alert">

<strong>Success</strong> @TempData["update\_message"]

<button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>

</div>

}

if (TempData["delete\_message"] != null)

{

<div class="alert alert-success alert-dismissible fade show" role="alert">

<strong>Success</strong> @TempData["delete\_message"]

<button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>

</div>

}

}

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.id)

</th>

<th>

@Html.DisplayNameFor(model => model.studentName)

</th>

<th>

@Html.DisplayNameFor(model => model.studentGender)

</th>

<th>

@Html.DisplayNameFor(model => model.age)

</th>

<th>

@Html.DisplayNameFor(model => model.standard)

</th>

<th>

@Html.DisplayNameFor(model => model.fatherName)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.id)

</td>

<td>

@Html.DisplayFor(modelItem => item.studentName)

</td>

<td>

@Html.DisplayFor(modelItem => item.studentGender)

</td>

<td>

@Html.DisplayFor(modelItem => item.age)

</td>

<td>

@Html.DisplayFor(modelItem => item.standard)

</td>

<td>

@Html.DisplayFor(modelItem => item.fatherName)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.id }) |

@Html.ActionLink("Details", "Details", new { id=item.id }) |

@Html.ActionLink("Delete", "Delete", new { id=item.id })

</td>

</tr>

}

</tbody>

</table>

* **Create.cshtml file:**

@model CRUDAppUsingASPCoreWebAPI6.Models.Student

@{

ViewData["Title"] = "Create";

}

<h1>Create</h1>

<h4>Student</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="id" class="control-label"></label>

<input asp-for="id" class="form-control" />

<span asp-validation-for="id" class="text-danger"></span>

</div> \*@

<div class="form-group">

<**label** **asp-for**="studentName" class="control-label"></**label**>

<**input** **asp-for**="studentName" class="form-control" />

<**span** **asp-validation-for**="studentName" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="studentGender" class="control-label"></**label**>

<**input** **asp-for**="studentGender" class="form-control" />

<**span** **asp-validation-for**="studentGender" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="age" class="control-label"></**label**>

<**input** **asp-for**="age" class="form-control" />

<**span** **asp-validation-for**="age" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="standard" class="control-label"></**label**>

<**input** **asp-for**="standard" class="form-control" />

<**span** **asp-validation-for**="standard" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="fatherName" class="control-label"></**label**>

<**input** **asp-for**="fatherName" class="form-control" />

<**span** **asp-validation-for**="fatherName" class="text-danger"></**span**>

</div>

<div class="form-group">

<input type="submit" value="Create" class="btn btn-primary" />

</div>

</**form**>

</div>

</div>

<div>

<**a** **asp-action**="Index">Back to List</**a**>

</div>

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

* **Edit.cshtml file:**

@model CRUDAppUsingASPCoreWebAPI6.Models.Student

@{

ViewData["Title"] = "Edit";

}

<h1>Edit</h1>

<h4>Student</h4>

<hr />

<div class="row">

<div class="col-md-4">

<**form** **asp-action**="Edit">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="id" class="control-label"></**label**>

<**input** **asp-for**="id" class="form-control" readonly />

<**span** **asp-validation-for**="id" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="studentName" class="control-label"></**label**>

<**input** **asp-for**="studentName" class="form-control" />

<**span** **asp-validation-for**="studentName" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="studentGender" class="control-label"></**label**>

<**input** **asp-for**="studentGender" class="form-control" />

<**span** **asp-validation-for**="studentGender" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="age" class="control-label"></**label**>

<**input** **asp-for**="age" class="form-control" />

<**span** **asp-validation-for**="age" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="standard" class="control-label"></**label**>

<**input** **asp-for**="standard" class="form-control" />

<**span** **asp-validation-for**="standard" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="fatherName" class="control-label"></**label**>

<**input** **asp-for**="fatherName" class="form-control" />

<**span** **asp-validation-for**="fatherName" class="text-danger"></**span**>

</div>

<div class="form-group">

<input type="submit" value="Save" class="btn btn-primary" />

</div>

</**form**>

</div>

</div>

<div>

<**a** **asp-action**="Index">Back to List</**a**>

</div>

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

* **Details.cshtml file:**

@model CRUDAppUsingASPCoreWebAPI6.Models.Student

@{

ViewData["Title"] = "Details";

}

<h1>Details</h1>

<div>

<h4>Student</h4>

<hr />

<dl class="row">

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.id)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.id)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.studentName)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.studentName)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.studentGender)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.studentGender)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.age)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.age)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.standard)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.standard)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.fatherName)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.fatherName)

</dd>

</dl>

</div>

<div>

@Html.ActionLink("Edit", "Edit", new { id = Model.id }) |

<**a** **asp-action**="Index">Back to List</**a**>

</div>

* **Delete.cshtml file:**

@model CRUDAppUsingASPCoreWebAPI6.Models.Student

@{

ViewData["Title"] = "Delete";

}

<h1>Delete</h1>

<h3>Are you sure you want to delete this?</h3>

<div>

<h4>Student</h4>

<hr />

<dl class="row">

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.id)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.id)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.studentName)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.studentName)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.studentGender)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.studentGender)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.age)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.age)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.standard)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.standard)

</dd>

<dt class = "col-sm-2">

@Html.DisplayNameFor(model => model.fatherName)

</dt>

<dd class = "col-sm-10">

@Html.DisplayFor(model => model.fatherName)

</dd>

</dl>

<**form** **asp-action**="Delete">

<input type="submit" value="Delete" class="btn btn-danger" /> |

<**a** **asp-action**="Index">Back to List</**a**>

</**form**>

</div>

* **In Program.cs fil change:**

app.MapControllerRoute(

name: "default",

pattern: "{controller=Student}/{action=Index}/{id?}");

**Identity Framework in ASP.NET Core 6:**

* ASP.NET Core Identity is a built-in membership system.
* It is used to Create, read, Update and Delete User accounts.
* Account Information.
* Authentication & Authorization
* Password Recovery
* Two factor authentication with SMS.
* It supports External Login Providers like Microsoft, Facebook, Google Email etc.
* ASP.NET Core provides identity membership system that enable us to add login functionality to our application.
* It is an API that supports User Interface (UI) Login Functionality.
* Manage users, password, profile data, roles, tokens, email confirmation.

**Registration/SignUp, login And Logout Using Identity in ASP.NET Core 6:**

**Step 1:**

* Add identity to our project.

Steps: Create a ASP.NET Core Web App (MVC) àName: ASPIdentityAppàRight Click on ASPIdentityApp project àClick on Addà Select New Scaffolded ItemàSelect IdentityàClick on AddàSelect an existing Layout page or specify a new one (Click on …(three dots)àGo to Viewsà SharedàSelect \_Layout.cshtmlàClick on OK

* Add Layout Page (Master Page).
* Select Features you want to add in Identity.
* For now, I want Account\Login, Account\Logout, Account\Register. So, I select these three.
* Create a Data Context Class which is used to communicate with our database.

Click on + (Plus) sign for Data context classàNew data context type give name “ApplicationDbContext”àClick on Add

* Create a User Class.

Click on + (Plus) sign for User classà Give name as “ApplicationUser”àClick on ADD

* After Adding Identity there is a folder called “Areas” appears.

Step 2:

* Add Register/Login Links in the Navbar.
* We have to call a “\_LoginPartial.cshtml” partial view in our Layout page by using partial tag.
* <**partial** **name**="\_LoginPartial"/>
* “LoginPartial.cshtml” partial view is located in Views/Shared folder, and that is added by identity.

Step 3:

* Add some properties in Application User class.
* Register these properties OR configure in ApplicationDbContext class.

Step 4:

* Now Add Connection String inside appsetting.json.
* Add Migration in package manager console.

Add-Migration MigrationName

* Update-Database

Step 5:

* Now Add TextBoxes for FirstName and LastName in Register Page.

**Dependency Injection:**

**What is Dependency Injection?**

* Dependency Injection (DI) is a design pattern used in software development,

Including .NET applications, to achieve loose coupling between components or classes.

* In simple terms, it allows you to separate the construction and usage of an object, making your code more maintainable, testable, and flexible.
* With DI, rather than creating objects directly inside a class, we provide the required dependencies from an external source, typically called an “IoC (Inversion of Control) container).
* The container is responsible for creating and managing instances of objects and injecting them into classes that need them.

**Why is Dependency Injection Needed?**

Dependency Injection is needed for several reasons:

* **Decoupling:** It reduces the direct dependencies between classes, making it easier to change or replace implementations without affecting other parts of the code.
* **Testability:** By Injecting dependencies, we can easily provide mock or fake objects during testing, allowing to isolate the behavior of individual components for more reliable and focused unit tests.
* **Flexibility**: DI enables you to configure different implementations of the dependency at runtime, providing flexibility in how our application behaves without changing its code.

**What happens if we do not use Dependency Injection?**

* Without dependency injection, classes tend to be tightly coupled, meaning they rely directly on concrete implementations of their dependencies. This makes it difficult to change implementations without modifying the dependent class.
* As a result, testing becomes more challenging, and the code becomes less maintainable and flexible.

**Pros and Cons of Dependency Injection:**

Pros:

* Improved maintainability and testability.
* Reduced coupling between classes.
* Better separation of concerns.
* Easier to introduce new features or modify existing behavior.]
* Better control over the lifecycle of objects.

Cons:

* Increased complexity due to the need for an IoC container.
* Potential performance overhead (though modern DI containers are highly optimized).

**How to implement Dependency Injection in ASP.NET Core 6 Web API CRUD Operations?**

* To implement dependency injection, follows these general steps.
* 1. Define your Services: Identify the various services (repositories, data access layers, business logic etc.) required to perform CRUD operations.
* Create Interfaces: Define Interfaces for each service that abstract their functionality.

The Program.cs Class:

* The Program.cs class is the top-level class in our application. The entire bootstrapping process of .NET Core application goes in the program.cs class.
* The process is like this:
* 1. Create a builder object from WebApplication.CreateBuilder()
* Add services to the Dependency Injection Service collection.
* Use the Build () method to create the app
* Add middleware.
* Run the app with the Run () method.

Steps:

1. Create web api project using .net 6.
2. Create a Student Class.
3. Create an interface and define a method.
4. Create a service and inherit the interface and implement the method.
5. Register the service in Program.cs.
6. Create a controller.
7. Inject dependency in controller and write a method to call the services method.
8. Test the application.

Code Part:

Create a project ExampleOfDependencyInjection:

1. Student.cs Class:

namespace ExampleOfDependencyInjection

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

}

}

1. IStudentRepo.cs class:

namespace ExampleOfDependencyInjection

{

public interface IStudentRepo

{

string PrintStudentName();

}

}

1. StudentRepo.cs:

namespace ExampleOfDependencyInjection

{

public class StudentRepo : IStudentRepo

{

public string PrintStudentName()

{

return "Keshav";

}

}

}

1. Program.cs class:

using ExampleOfDependencyInjection;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddScoped<IStudentRepo, StudentRepo>();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. StudentController.cs class:

using Microsoft.AspNetCore.Mvc;

// For more information on enabling Web API for empty projects, visit https://go.microsoft.com/fwlink/?LinkID=397860

namespace ExampleOfDependencyInjection

{

[Route("api/[controller]")]

[ApiController]

public class StudentController : ControllerBase

{

private readonly IStudentRepo \_studentRepo;

public StudentController(IStudentRepo studentRepo)

{

\_studentRepo = studentRepo;

}

// GET: api/<StudentController>

[HttpGet]

public string Get()

{

return \_studentRepo.PrintStudentName();

}

// GET api/<StudentController>/5

[HttpGet("{id}")]

public string Get(int id)

{

return "value";

}

// POST api/<StudentController>

[HttpPost]

public void Post([FromBody] string value)

{

}

// PUT api/<StudentController>/5

[HttpPut("{id}")]

public void Put(int id, [FromBody] string value)

{

}

// DELETE api/<StudentController>/5

[HttpDelete("{id}")]

public void Delete(int id)

{

}

}

}

**Why do we need service lifetimes in dependency injection in .NET Core?**

* In the context of Dependency Injection service lifetimes are essential for managing the lifecycle and behavior of the services or objects that get injected into classes.
* Service lifetimes dictate how long an instance of a service should exist and when it should be disposed of.

**Registration Lifetimes:**

* When we register services to the DI services container, we must specify one of three lifetime options, and then pass in the interface as the first parameter and the actual class as the second parameter.
* When we register our services, we have three options:
* 1. Singleton
* 2. Scoped
* 3. Transient

1. Singleton:

* Singleton means that an instance of the service will run from the moment the application starts until shut down.
* Something that might use singleton is perhaps a logger, a task scheduler, or something that needs to be running in the background all the time.
* This one instance could be running for days or months at a time.
* It is the common registration type of the three.

1. Scoped:

* Scoped means that the app will create a new instance of the server for each API request.
* In other words, that is short, probably measured in the milliseconds.
* However, if in a single request, the service is invoked than once, the app will use the same instance.

1. Transient:

* Transient is similar to scoped but it is lifetime is even shorter.
* With transient, the app will create a new instance of the class anytime its invoked.
* This is the most common registration type and the one we should use with CRUD style services.
* Don’t forget that our application could be handling hundreds or thousands of requests every minute, so being able to specify how long resources are allocated to services provides great control on making efficient apps.

**Summary:**

* With a transient service, a new instance is provided every time an instance is requested whether it is in the scope of same HTTP request or across different HTTP requests.
* With a scoped service we get the same instance within the scope of a given HTTP request but a new instance across different HTTP requests.
* With Singleton service, there is only a single instance. An instance is created, when service is first requested, and that single instance will be used by all subsequent HTTP request throughout application.

**Steps:**

1. Create a web api project.
2. Create three interface and define a method to print the data ticks.
3. Create three services and inherit the interface and implement the method.
4. Register the service in program.cs with all three service lifetimes.
5. Create a Controller and inject two instances of each service.
6. Call the service function from the controller.
7. Create a web api project name as DependencyInjectionAllThreeServices
8. Create Interfaces folder inside this create 3 interfaces.

ISingleton.cs :

namespace DependencyInjectionAllThreeService.Interfaces

{

public interface ISingleton

{

string PrintGuidNumber();

}

}

IScoped.cs:

namespace DependencyInjectionAllThreeService.Interfaces

{

public interface IScoped

{

string PrintGuidNumber();

}

}

ITransient.cs:

namespace DependencyInjectionAllThreeService.Interfaces

{

public interface ITransient

{

string PrintGuidNumber();

}

}

1. Create a Services folder where we create class which implement method of these interfaces.

Singleton.cs class:

using DependencyInjectionAllThreeService.Interfaces;

namespace DependencyInjectionAllThreeService.Services

{

public class Singletons : ISingleton

{

private readonly Guid number;

public Singletons()

{

number = Guid.NewGuid();

}

public string PrintGuidNumber()

{

return number.ToString();

}

}

}

Scoped.cs class:

using DependencyInjectionAllThreeService.Interfaces;

namespace DependencyInjectionAllThreeService.Services

{

public class Scoped : IScoped

{

private readonly Guid number;

public Scoped()

{

number = Guid.NewGuid();

}

public string PrintGuidNumber()

{

return number.ToString();

}

}

}

Transient.cs class:

using DependencyInjectionAllThreeService.Interfaces;

namespace DependencyInjectionAllThreeService.Services

{

public class Transient : ITransient

{

private readonly Guid number;

public Transient()

{

number= Guid.NewGuid();

}

public string PrintGuidNumber()

{

return number.ToString();

}

}

}

1. Program.cs class add services here.

using DependencyInjectionAllThreeService.Interfaces;

using DependencyInjectionAllThreeService.Services;

namespace DependencyInjectionAllThreeService

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddSingleton<ISingleton,Singletons>();

builder.Services.AddTransient<ITransient,Transient>();

builder.Services.AddScoped<IScoped, Scoped>();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}

1. Add a model class name as DateResponse.cs class.

namespace DependencyInjectionAllThreeService

{

public class DateResponse

{

public string Singleton { get; set; }

public string Singleton1 { get; set; }

public string Transient { get; set;}

public string Transient1 { get; set;}

public string Scoped { get; set;}

public string Scoped1 { get; set;}

}

}

1. Create a controller class name as DateController.cs.

using DependencyInjectionAllThreeService.Interfaces;

using Microsoft.AspNetCore.Mvc;

namespace DependencyInjectionAllThreeService.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class DateController : ControllerBase

{

private readonly ISingleton singleton;

private readonly ISingleton singleton1;

private readonly ITransient transient;

private readonly ITransient transient1;

private readonly IScoped scoped;

private readonly IScoped scoped1;

public DateController(ISingleton singleton,ISingleton singleton1,ITransient transient ,ITransient transient1,

IScoped scoped,IScoped scoped1)

{

this.singleton = singleton;

this.singleton1 = singleton1;

this.transient = transient;

this.transient1 = transient1;

this.scoped = scoped;

this.scoped1 = scoped1;

}

//GET: api/<DateController>

[HttpGet]

public DateResponse Get()

{

var response = new DateResponse();

response.Singleton = singleton.PrintGuidNumber();

response.Singleton1 = singleton1.PrintGuidNumber();

response.Transient = transient.PrintGuidNumber();

response.Transient1 = transient1.PrintGuidNumber();

response.Scoped = scoped.PrintGuidNumber();

response.Scoped1 = scoped1.PrintGuidNumber();

return response;

}

//public IActionResult Index()

//{

// return View();

//}

}

}

OUTPUT:

A screenshot of a computer program

Description automatically generated

**The Big Picture of JWT Authentication:**

* Before we get into implementation of authentication and authorization, lets have a quick look at the JWT authentication big picture.
* There is an application that has a login form. A user enters their username, and password and presses the login button. After pressing the login button, a client (e.g web browser sends the user’s data to the server’s API endpoint).
* When the server validates the user’s credentials and confirmation the user is valid, it’s going to send an encoded JWT to the client.
* JSON web token is basically a JavaScript object that can some attributes of the logged-in user. It can contain a username, user subject, user role or some other useful information.

What is JWT (JSON Web Token)?

* JSON web tokens enable a secure way to transmit data between two parties in the form of JSON object. It is an open standard, and it is popular mechanism for web authentication. In our case, we are going to use JSON web tokens to securely transfer a user’s data between the client and the server.
* JSON web tokens consist of three basic parts: the header, payload, and signature.

Header:

* The first part of JWT is the header, which is a JSON object encoded in the base64 format.
* The header is a standard part of JWT and we don’t have to worry about it.
* It contains information like the type of token and the name of the algoritm:

{“alg”: “HS256”,”typ”:”JWT”}

Payload:

* After the Header, we have a Payload which is also a JavaScript object encoded in the base64 format.
* The payload contains some attributes about the logged-in user. For example, it can contain the user id, user subject, and information about whether a user is an admin user or not.
* JSON web tokens are not encrypted. It can be decoded with any base64 decoder so we should never include sensitive information in the Payload.
* {

“sub”:”12345”,

“name”:”John Doe”,

“iat”:123639859

}

Signature:

* Finally, we have the Signature part. Usually, the server uses the signature part to verify whether the token contains valid information- the information the server is issuing.
* It is a digital signature that gets generated by combining the header and the payload together.
* Moreover, it is based on a secret key that the server knows:

PRACTICAL:

* Create ASP.NET Core Web API Project
* Step 1:
* Install the required packages from Nuget Package Manager.
* 1. Microsoft.AspNetCore.Authentication.JWTBearer
* 2. Microsoft.EntityFrameworkCore.SqlServer
* 3. Microsoft.EntityFrameworkCore.Tools
* 4. Microsoft.EntityFrameworkCore
* Step 2:
* Create Model folder and create two class.

1. User and 2. Employee

Login Action Explanation:

* We decorate our Login action with the HttpPost attribute.
* Inside the login method, we create the SymmetricSecretKey with the secret key value superSecretKey@345.
* Then, we create the SigningCredentials object and as arguments, we provide a secret key and the name of the algorithm that we are going to use to encode the token.

We create JwtSecurityToken object with some important parameters:

* Issuer: The first parameter is a simple string representing the name of the webserver that issues the token.
* Audience: The second parameter is a string value representing the valid recipients.
* Claims: The third argument is a list of user roles, for example, the user can be admin, manager.
* Expires: The fourth argument is the DateTime object that represents the date and time after which the token expires then, we create a string representation of JWT by calling the WriteToken method on JwtSecurityTokenHandler.
* Finally, we return JWT in a response. As a response, we create the Authenticatedresponse object that contains only the Token property.
* In Program.cs class we have to configure JWT:

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme

).AddJwtBearer(options=>

{

options.RequireHttpsMetadata = false;

options.SaveToken = true;

options.TokenValidationParameters = new Microsoft.IdentityModel.Tokens.TokenValidationParameters()

{

ValidateIssuer = true,

ValidateAudience = true,

ValidAudience = builder.Configuration["Jwt:Audience"],

ValidIssuer = builder.Configuration["Jwt:Issuer"],

IssuerSigningKey = new

SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration

["Jwt:Key"]))

};

});

* In appsettings.json

"jwt": {

"Key": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNDU2Nzg5LCJuYW1lIjoiSm9zZXBoIn0.OpOSSw7e485LOP5PrzScxHb7SR6sAOMRckfFwi4rp7o",

"Issuer": "JWTAuthenticationServer",

"Audience": "JWTServicePostmanClient",

"Subject": "JWTServiceAccessToken"

}

**LINQ FIRST Method in C#:**

* The LINQ First Method in C# is used to return the first element from a data source or from a collection. If the data source or collection is empty, or if we specified a condition and with that condition, no element is found in the data source, then the LINQ First method will throw an InvalidOperationException.
* If the data source is Null, then it will throw ArgumentNullException.

Example:

First\_FirstOrDefault.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace First\_FirstOrDefault\_Single\_SingleOrDefault

{

internal class First\_FristOrDefault

{

public void FirstExample()

{

List<int> numbers = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

//Using Method Syntax

int methodSyntax = numbers.First();

//Using Query Syntax

int QuerySyntax = (from num in numbers

select num).First();

//printing the value returned by the First Method

Console.WriteLine(methodSyntax);

}

public void FirstWithPredicate()

{

//Fetching the first element from the data source which is divisible by 2

List<int> numbers = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

//Using Method Syntax

int methodSyntax = numbers.First(num => num % 2 == 0);

//Using Query Syntax

int querySyntax = (from num in numbers

select num).First(num => num % 2 == 0);

//Printing the value returned by the First Method

Console.WriteLine(methodSyntax);

}

public void FirstWithNullDataSource()

{

List<int> numberEmpty = new List<int>() { };

int methodSyntax = numberEmpty.First();

Console.WriteLine(methodSyntax);

}

public void FirstWithCollectionNotReturningData()

{

List<int> numbers = new List<int>() {1,2,3,4,5,6,7,8,9,10 };

int methodSyntax = numbers.First(num => num > 50);

Console.WriteLine(methodSyntax);

}

}

}

Program.cs:

using First\_FirstOrDefault\_Single\_SingleOrDefault;

internal class FirstandFirstOrDefault

{

static void Main(string[] args)

{

First\_FristOrDefault first\_FristOrDefault = new First\_FristOrDefault();

first\_FristOrDefault.FirstExample();

first\_FristOrDefault.FirstWithPredicate();

first\_FristOrDefault.FirstWithNullDataSource();

first\_FristOrDefault.FirstWithCollectionNotReturningData();

}

}

**LINQ FirstOrDefault Method:**

* The LINQ FirstOrDefault Method in C# exactly does the same thing as the LINQ First method except that this method does not throw the InvalidOperationException exception when the data source is empty or when the specified condition does not match with any element in the data source. In such cases, it will return the default value based on the data type of the data source.
* If the Data Source is Null, then like the First method, it will also throw ArgumentNullException.

namespace FirstOrDefaultDemo

{

public class Program

{

public void FirstOrDefaultExample()

{

List<int> numbers = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int methodSyntax = numbers.FirstOrDefault();

Console.WriteLine(methodSyntax);

int querySyntax= (from num in numbers

select num).FirstOrDefault();

Console.WriteLine(querySyntax);

}

public void FirstOrDefaultExample1()

{

List<int> numbers = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int methodSyntax = numbers.FirstOrDefault(num=>num%2==0);

Console.WriteLine(methodSyntax);

int querySyntax= (from num in numbers

select num).FirstOrDefault(num=> num%2==0);

Console.WriteLine(querySyntax);

}

public void FirstOrDefaultExample2()

{

List<int> numbers = new List<int>() {1,2,3,4,5,6,7,8,9,10 };

int methodSyntax = numbers.FirstOrDefault(num => num > 50);

Console.WriteLine(methodSyntax);

int querySyntax = (from num in numbers

select num).FirstOrDefault(num => num > 50);

Console.WriteLine(querySyntax);

}

public void FirstOrDefaultExample3()

{

List<int> numbers = new List<int>() { };

int methodSyntax = numbers.FirstOrDefault();

Console.WriteLine(methodSyntax);

int querySyntax = (from num in numbers

select num).FirstOrDefault();

Console.WriteLine(querySyntax);

}

static void Main(string[] args)

{

Program p = new Program();

p.FirstOrDefaultExample();

p.FirstOrDefaultExample1();

p.FirstOrDefaultExample2();

p.FirstOrDefaultExample3();

}

}

}

**LINQ Single Method in C#:**

* The LINQ Single Method is used to return a single element from a data source or you can say from a sequence.
* Public static TSource Single<TSource> (this IEnumerable<TSource> source);
* Public static TSource Single<TSource>(this IEnumerable<TSource> source, Func<TSource,bool> predicate);
* The first overloaded version of the Single method does not take any parameter. This method simply returns the only element from a sequence.
* In this case, if the data source is empty or if the data source contains more than one element, then it throws an exception.
* On the other hand, the second overloaded version of the Single method takes one predicate as a parameter and using this predicate we can specify a condition.
* This method returns the only element from the sequence which satisfied the given condition.
* In this case, the method will throw an exception when any of the following conditions is true.
* 1. If the data source is empty.
* 2. When the given condition does not satisfy any element in the sequence.
* 3. If the given condition satisfies more than one element.

LINQ Single Method with Empty Data Source:

* When we apply the LINQ Single Method on the empty sata source then it will throw System.InvalidOperationException: ‘Sequence contains no element’ exception.

What happens when the Sequence contains more than one element?

* If the Sequence contains more than one element and if we apply the Single method on that sequence, then we will get System.InvalidOperationException: ‘Sequence contains more than one element’.

namespace SingleLinqExample

{

public class Program

{

public void SingleExample()

{

List<int> numbers = new List<int>() { 10 };

int numberMS = numbers.Single();

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).Single();

Console.WriteLine(numberQS);

}

//public void SingleWithEmptyDataSource()

//{

// List<int> numbers= new List<int>() { };

// int numberMS = numbers.Single();

// Console.WriteLine(numberMS);

// int numberQS = (from num in numbers

// select num).Single();

// Console.WriteLine(numberQS);

//}

//public void SingleWithMoreThanOneElement()

//{

// List<int> numbers = new List<int>() { 10, 20, 30 };

// int numberMS = numbers.Single();

// Console.WriteLine(numberMS);

// int numberQS = (from num in numbers

// select num).Single();

// Console.WriteLine(numberQS);

//}

public void SingleWithPredicate()

{

List<int> numbers = new List<int> { 10, 20, 30 };

int numberMS = numbers.Single(num=> num==20);

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).Single(num=>num==20);

Console.WriteLine(numberQS);

}

public void SingleIfConditionReturnsMoreThanOneValue()

{

List<int> numbers = new List<int>() { 10, 20, 30 };

int numberMS = numbers.Single(num => num > 10);

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).Single(num=>num>10);

Console.WriteLine(numberQS);

}

static void Main(string[] args)

{

Program p = new Program();

p.SingleExample();

//p.SingleWithEmptyDataSource();

// p.SingleWithMoreThanOneElement();

p.SingleWithPredicate();

p.SingleIfConditionReturnsMoreThanOneValue();

}

}

}

**LINQ SingleOrDefault Method in C#:**

* The LINQ SingleOrDefault method is very much similar to the LINQ Single method except that this method will not throw an exception when the sequence is empty or when no element in the sequence satisfied the given condition.
* This method returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies a specified condition.

using System.Linq;

namespace SingleOrDefaultExample

{

public class Program

{

public void SingleOrDefaultExample()

{

List<int> numbers = new List<int>() { 10 };

int numberMS = numbers.SingleOrDefault();

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).SingleOrDefault();

Console.WriteLine(numberQS);

}

public void SingleOrDefaultWithEmptyDataSource()

{

List<int> numbers = new List<int>() { };

int numberMS = numbers.SingleOrDefault();

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).SingleOrDefault();

Console.WriteLine(numberQS);

}

public void SingleOrDefaultWithMoreThanOneElement()

{

List<int> numbers = new List<int>() { 10, 20, 30 };

int numberMS = numbers.SingleOrDefault();

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).SingleOrDefault();

Console.WriteLine(numberQS);

}

public void SingleWithPredicate()

{

List<int> numbers = new List<int> { 10, 20, 30 };

int numberMS = numbers.SingleOrDefault(num => num == 20);

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).SingleOrDefault(num => num == 20);

Console.WriteLine(numberQS);

}

public void SingleOrDefaultIfConditionReturnsMoreThanOneValue()

{

List<int> numbers = new List<int>() { 10, 20, 30 };

int numberMS = numbers.SingleOrDefault(num => num > 10);

Console.WriteLine(numberMS);

int numberQS = (from num in numbers

select num).SingleOrDefault(num => num > 10);

Console.WriteLine(numberQS);

}

static void Main(string[] args)

{

Program p = new Program();

p.SingleOrDefaultExample();

p.SingleOrDefaultWithEmptyDataSource();

p.SingleOrDefaultWithMoreThanOneElement();

p.SingleOrDefaultIfConditionReturnsMoreThanOneValue();

}

}

}

**SOLID PRINCIPLE:**

**S- Single Responsibility:**

What is Single Responsibility Principle?

* Single Responsibility Principle (SRP) states that a class should have only one responsibility.
* Or a class should have only one reason to change.
* When a class has only one responsibility, it becomes easier to change and test. If a class has multiple responsibilities, changing one responsibility may impact others and more testing efforts will be required then.

**O – Open-closed Principle**

What is Open-Closed Principle?

* Open-Closed Principle (OCP) states that software entities (classes, modules) should be open for extension, but closed for modification.
* SRP is the prerequisite for OCP.
* The benefit is simple testing is required to test individual classes, but if you will keep on adding and modifying in one class. Then even for the smallest modification, the whole class needs to be tested.

**L- Liskov Substitution Principle**

What is Liskov Substitution Principle?

* The Liskov Substitution Principle (LSP) states that an object of a child class must be able to replace an object of the parent class without breaking the application.
* All the base class method must be applicable for the derived class.

**I – Interface Segregation Principle**

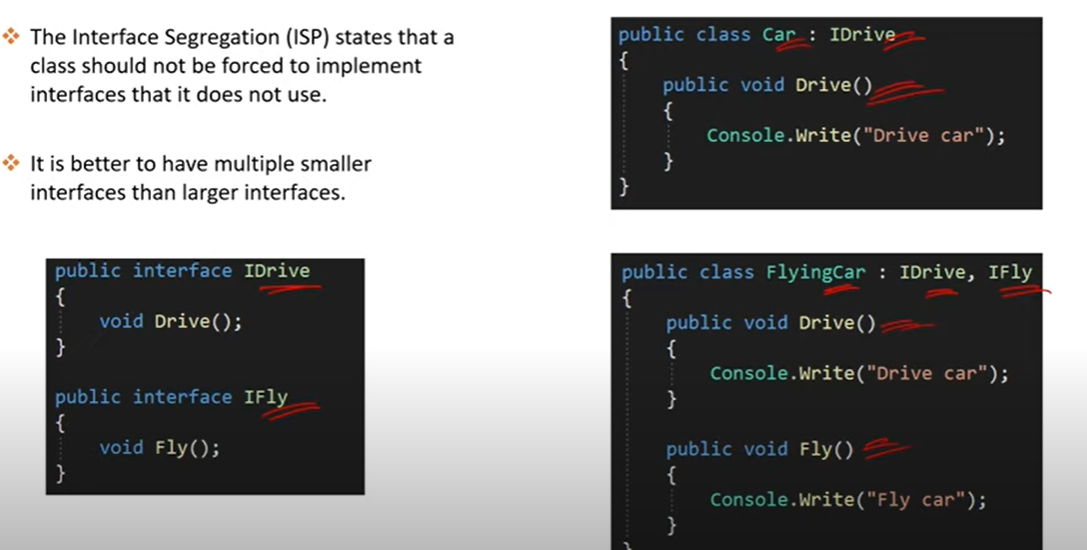
What is Interface Segregation Principle?

* The Interface Segregation (ISP) states that a class should not be forced to implement interfaces that it does not use.
* It is better to have multiple smaller interfaces than large interfaces.

**D – Dependency Inversion Principle**

What is Dependency Inversion Principle?

* The Dependency Inversion Principle (DIP) states that a high-level class must not depend upon a lower-level class.

****

**Design Patterns:**

* The Design patterns are simply the reuseable solutions to the problems which we are facing every day in our programming.
* Design patterns are generally used for solving the problems of object creation and implementation.
* These design patterns are basically treated as templates that can be applied to the real-world problems.
* There are 23 design patterns. These 23 patterns are grouped into three main categories.

1. Creational Design Pattern
2. Structural Design Patterns
3. Behavioral Design Patterns
4. Creational Design Pattern

* Creational Design Patterns focus on the process of object creation in software development.
* These patterns make sure that we create things in a way that’s not only easy but easy but also flexible, so we can change them later if we need too. They hide the complicated details of how we put pieces together.
* This type of pattern deals with the generation and initialization of object.

Singleton

Factory Method

Abstract Factory

Prototype

Builder

1. Structural Design Patterns

* The design patterns in this category deals with the class structure such as Inheritance and Composition.
* This type of pattern deals with the creation of class and object.
* This type of pattern deals with the Interface decoupling.

Façade

Decorator

Composite

Adaptor

Bridge

Flyweight

Proxy

1. Behavioral Design Patterns:

* This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.
* This type of design explains the connection between classes and their objects.
* This type of design deals with the communication between classes and objects.

Command

Interpreter

Chain of Responsibility

Iterator

Observer

Mediator

Memento

State

Visitor

Template Method

Strategy

**Singleton Pattern:**

What is a Singleton Pattern?

* If we want a class to have only one object, then we use the Singleton Pattern.

Why do we need only one object of a class?

* Objects used for logging, device drivers for printers or registry settings need to have only one of a kind. If there are multiple objects we may encounter problems like incorrect program behavior, inconsistent results etc.

Another Class 1

Request

MyClass

Single Instance

Another Class 2

Request

* Singleton design patterns in C# is one of the most popular design patterns.
* In this pattern, a class has only one instance in the program that provides a global point of access to it.
* In other words, a singleton is a class that allows only a single instance of itself to be created and usually gives simple access to that instance.
* The Single instance will remain same throughout the lifetime of the application.

Why do we need Singleton Design Pattern?

* When there is single resource throughout the application.

For example: database, log file etc.

* When we want to pass instance from one class to another class.

Implementation:

* Class should be sealed.
* Create a private objects default constructor to prevent other classes from creating objects using a new keyword with the singleton class.
* Create a static method that acts as a constructor.

Example:

* Create a console project name it SingletonDemo. Inside this project create a class name as MyClass.cs. Now in MyClass.cs class write some code for achieve singleton pattern.
* This pattern is called Double-Check Locking.

Code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

sealed class MyClass

{

private MyClass() { }

private static MyClass instance;

private static object instanceLock = new object();

public static MyClass GetInstance()

{

if (instance == null)

{

lock (instanceLock)

{

if (instance == null)

{

instance = new MyClass();

}

}

}

return instance;

}

}

}

* Inside main method that means in Program.cs class:

namespace SingletonDemo

{

internal class Program

{

static void Main(string[] args)

{

MyClass obj = MyClass.GetInstance();

MyClass obj2 = MyClass.GetInstance();

}

}

}

**Why we use this double-check locking?(Hindi)**

Singleton hmesha single instance create krta hai through out application. But jab hm multithread use krte hain to parallelly multiple thread ek saath kaam krta hai. To jab multithread rhega to wo singleton mai separate instance create krega ek time pe. Suppose ki ek time pe 2 thread aayega to wo if condition ke ander check krega ki instance null hai toh dono thread ke liye ye condition true ho jaayega jisse ki 2 instance create ho jaayega.

if (instance == null)

{

instance = new MyClass();

}

* jo ki singleton design pattern ka rule destroy hogya to wsse bachne ke liye hmlog lock-checking ka use krte hai.
* Lock checking mai kya hota ki at a time single thread hi enter krega rest of thread wait krega jab tak ki pehla thread kaam na kar le.
* Pehle hm log ek object create krte hain jo ki object type ka hoga ye parent hai saari class ki.
* private static object instanceLock = new object();
* Ab MyClass ke ander hm lock create krege. Ab isme jab first thread aayega aur wo condition check krega instance==null hai ki nhi ye null hoga first thread ke liye to wo ek instance create kr dega but jab second thread aayega tab condition fail ho jaayega aur wo simply pehle wala instance ko return kr dega.

lock (instanceLock)

{

if (instance == null)

{ instance = new MyClass(); } }

* Ab yha bhi ek problem aayega wo ye ki lock (instanceLock) ye pe bhut sara thread wait krega jisse ki performance issue aayega application bhut slow jaayega to wsse bachne ke liye ek aur if condition lgayega jo check krega instance null hai ya nhi isi ko Double-checking lock kehte hain.
* Ab Jab pehle if condition mai jaayega tab wo ek instance create kar dega but ab jab second thread jaayega to wo condition hi false ho jaayega aur direct created instance ko return kar dega multiple thread ko lock pe wait krne ka jarurat hi nhi prega jisse ki performance issue bhi solve ho jaayega.

if (instance == null)

{

lock (instanceLock)

{

if (instance == null)

{

instance = new MyClass();

}

}

}

return instance;

**Factory Design Pattern:**

* This pattern comes under the creational patterns which are widely used in programming constructs.
* In other words, this pattern provides the best way to create the Objects.
* In Factory patterns we create the object of the Class without Exposing the Creation/Instantiation Logic to the user who wants to create the Object and then return the newly Created object using the Common Interface which is inherited by the Class.
* In simple words, if we have a superclass and n number of subclasses, and based on the data provided, if we have to create and return the object of one of the subclasses, then we need to use the Factory Design Pattern in C#.

Example:

* Create a console project name is FactoryDesign. Inside this project now we are going to create interface IVehicle.cs which have two method VehicleType() and NumberOfWheels (). Now for that method we have to create two class name is Bike.cs and Car.cs class which will be implement that interface method.
* Now for creating object of that classes we are going to create a class VehicleFactory.cs. This class have a method with one parameter and will create object of class based on the user input.
* Now in the Program.cs class inside the main method we are going to write code for user input and call the particular class.

Codes:

IVehicle.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryDesign

{

internal interface IVehicle

{

string VehicleType();

int NumberOfWheels();

}

}

Bike.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryDesign

{

internal class Bike : IVehicle

{

private readonly int \_wheels;

public Bike()

{

this.\_wheels = 2;

}

public int NumberOfWheels()

{

return this.\_wheels;

}

public string VehicleType()

{

return "Bike";

}

}

}

Car.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryDesign

{

internal class Car : IVehicle

{

private readonly int \_wheels;

public Car()

{

this.\_wheels = 4;

}

public int NumberOfWheels()

{

return this.\_wheels;

}

public string VehicleType()

{

return "Car";

}

}

}

VehicleFactory.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryDesign

{

internal class VehicleFactory

{

public static IVehicle GetVehicle(string objType)

{

IVehicle objectType=null;

if (objType.ToLower().Equals("bike"))

{

objectType = new Bike();

}

else if (objType.ToLower().Equals("car"))

{

objectType = new Car();

}

return objectType;

}

}

}

Program.cs:

namespace FactoryDesign

{

internal class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter Vehicle Type");

string userInput = Console.ReadLine();//bike//car

IVehicle Type = VehicleFactory.GetVehicle(userInput);

Console.WriteLine("Vehicle Type is : " + Type.VehicleType());

Console.WriteLine("Number of Wheels : "+ Type.NumberOfWheels());

Console.ReadLine();

}

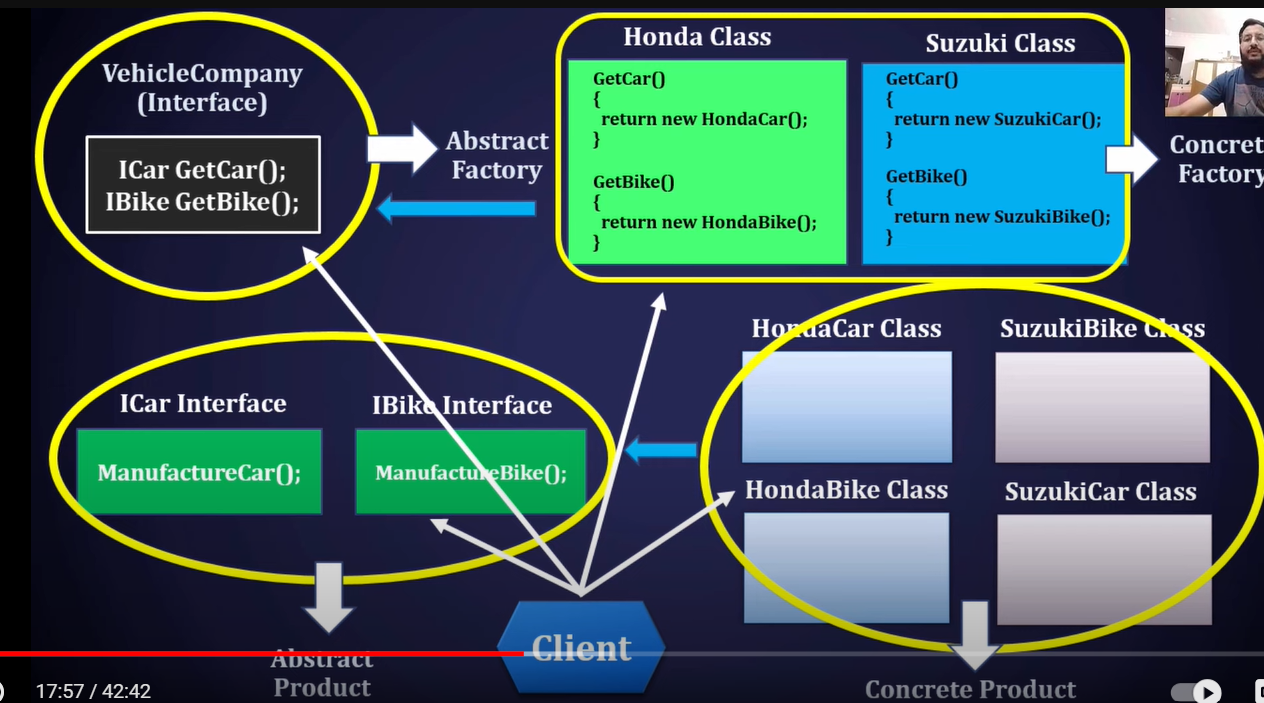
}

}

**Abstract Factory Design Pattern:**

* The Abstract Factory Design Pattern in C# is a creational pattern that provides an interface for creating product families of related or dependent objects without specifying their concrete classes.
* It’s part of the Gang of Four design patterns and is particularly useful when a system needs to be independent of how its products are created, composed, and represented.
* An Abstract Factory is like a super factory that creates other factories (Factory of Factories).
* Here’s overview of the components involved in the Abstract Factory pattern:

1. AbstractFactory
2. ConcreteFactory
3. AbstractProduct
4. ConcreteProduct
5. Client



* AbstractFactory: An interface with methods for creating abstract products.
* ConcreteFactory: Classes that implement the AbstractFactory interface and provide implementations for the interface methods to create concrete products objects.
* AbstractProduct: Interfaces for a type of product object, defining the operations a product should have.
* ConcreteProduct: Classes that implement the AbstractProduct interface.
* Client: Uses interfaces declared by AbstractFactory and AbstractProduct classes to create a family of products.

Example Code:

Create a Console Project.

namespace AbstractFactoryPattern

{

interface ICar

{

void ManufactureCar();

}

interface IBike

{

void ManuFactureBike();

}

class HondaCar : ICar

{

public void ManufactureCar()

{

Console.WriteLine("Honda Car is Manufactured.");

}

}

class HondaBike : IBike

{

public void ManuFactureBike()

{

Console.WriteLine("Honda Bike is Manufactured.");

}

}

class SuzukiCar : ICar

{

public void ManufactureCar()

{

Console.WriteLine("Suzuki Car is Manufactured.");

}

}

class SuzukiBike : IBike

{

public void ManuFactureBike()

{

Console.WriteLine("Suzuki Bike is Manufactured.");

}

}

interface VehicleCompany

{

ICar GetCar();

IBike GetBike();

}

class Honda : VehicleCompany

{

public IBike GetBike()

{

return new HondaBike();

}

public ICar GetCar()

{

return new HondaCar();

}

}

class Suzuki : VehicleCompany

{

public IBike GetBike()

{

return new SuzukiBike();

}

public ICar GetCar()

{

return new SuzukiCar();

}

}

internal class Program

{

static void Main(string[] args)

{

VehicleCompany hondaCompany = new Honda();

ICar hondaCar = hondaCompany.GetCar(); // new HondaCar;

IBike hondaBike = hondaCompany.GetBike();//new HondaBike();

VehicleCompany suzukiCompany = new Suzuki();

ICar suzukiCar = suzukiCompany.GetCar();

IBike suzukiBike = suzukiCompany.GetBike();

hondaCar.ManufactureCar();

hondaBike.ManuFactureBike();

suzukiCar.ManufactureCar();

suzukiBike.ManuFactureBike();

Console.ReadLine();

}

}

}