ASP.NET CORE 6

&

ENTITY FRAMEWORK CORE

&

ASP.NET CORE WEB API

&

IDENTITY FRAMEWORK

-: Pawan Kumar Vishwakarma

ASP .NET CORE 6

(1) What is .Net Core ?

- .Net Core is a framework for building web Apps, Mobile & Desktop Applications.
- It is free & open source.
- Developed by microsoft.

(2) What is .Net standard ?

- .Net standard defines a set of rules, which if any base class library of any framework will follow that the framework will be called .NET standard compliant.

(3) What are the advantages of .NET Core over .NET framework ?

⇒ Cross Platform

.NET framework supports only windows.

.NET core framework supports windows, linux, MacOS.

⇒ Open Source

.NET is paid.

.NET core is free to use.

⇒ Performance

.NET performance is good.

.NET core performace is faster than .NET.

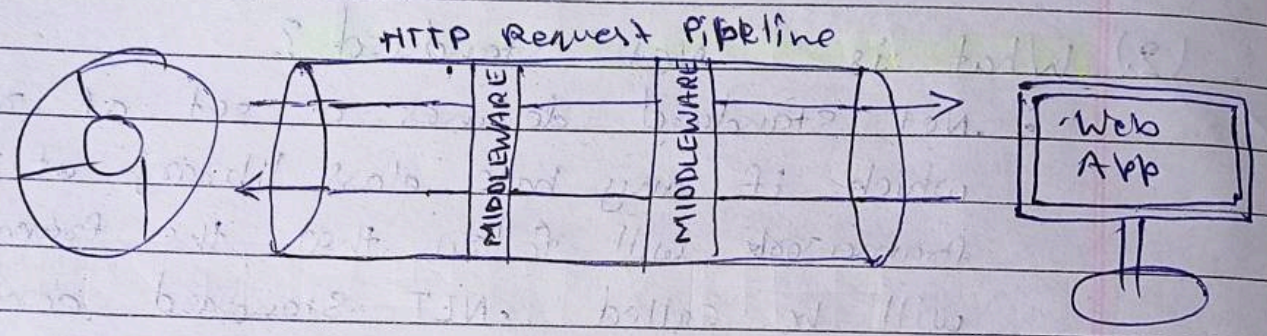
⇒ Language Support

.NET supports C#, F# & VB

.NET supports C# and F#

Middleware:

- It is the simple classes.
- It will be present in the program.cs file.
- Every request & response will go through the middleware.
- It runs on the ~~time~~ every request.



* 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.

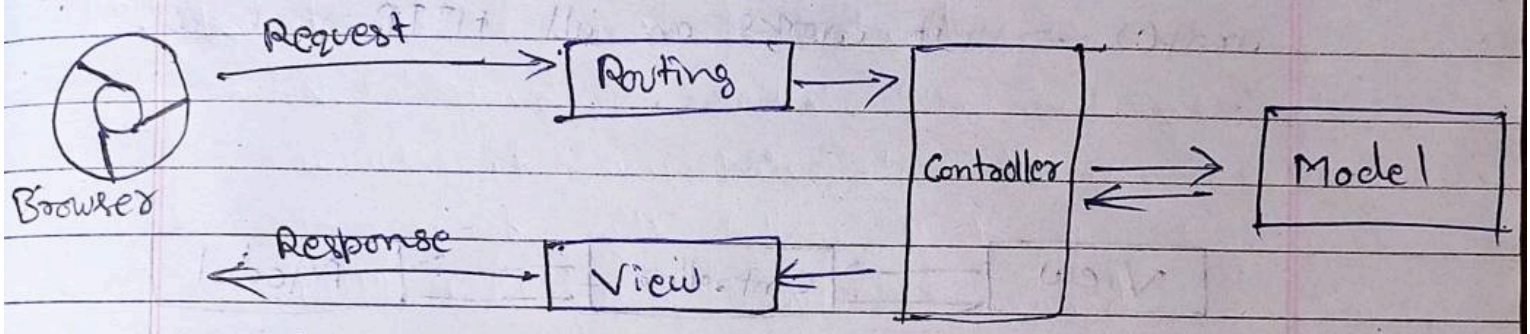
* Middleware are s/w components that are assembled into an application pipeline (program.cs) to handle requests and responses.

* It can also control how our application looks when there is an error.

* It is a key piece in how we authenticate and authorize a user.

⇒ Routing:

It is a mechanism in which it will ~~rout~~ inspect the incoming requests and then map that request to the controllers and their action methods.



Routing = URL + HTTP Methods { GET, POST, PUT, DELETE }

* There are two types of Routing:-
  (1) Convention-Based Routing → we define routing Pattern
  (2) Attribute-Based Routing → we use Route[] attribute.

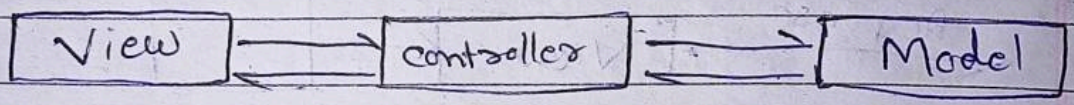
- We can use both Routing in single application.

* In Attribute-Based Routing
  - Token for controller is [controller]
  - Token for action method is [action]
  - We can use both tokens together [controller]/[action]

* Routing Methods :

- Map
- MapGet
- MapPost
- MapPut
- MapDelete

.map() → it works on all HTTP methods.



⇒ Controller.

- Controller manages the flow of the application.
- A controller is used to define and group a set of actions.
- It is responsible for intercepting incoming requests and executing the appropriate app. code.

- Controller is a backbone of MVC.
- Controller 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.
- It is recommended that class name of a controller ends with suffix 'Controller'.
- Locate the appropriate method to call for an incoming request.
- Validate the data of the incoming request before invoking the request method.
- Retrieve the request data and passing it to requested method as arguments.

⇒ Action Methods

- A controller class can contains 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.

⇒ Example of Action methods with Routing (convention based)

Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
var app = builder.Build();
app.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}"
        );
app.Run();
```

HomeController.cs

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

- Action methods are responsible for processing the requests that are sent to the controller.
- Action method can return multiple types of data. ex. ViewResult, PartialViewResult, String, int etc. Json
- By Default, it generates a response in the form of IActionResult interface or ActionResult Abstract class.
- Action method must be public. It can not be private or protected. They can not be declared as static.

- All the public methods inside a controller which respond to the URL are known as Action Methods.

- In an ASP.NET MVC application, you can create multiple action methods in a controller.

- We can invoke an action method by specifying a URL in the web browser containing the name of the controller and the action method to invoke.

⇒ <u>IActionResult:</u> (interface)

- The IActionResult return type is appropriate when multiple ActionResult return types are possible in an action.

- IActionResult and ActionResult work 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 response.
ViewResult	Represents an ActionResult that renders a view to the response.
ViewComponentResult	An IActionResult which renders a view component to the response.

⇒ 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 both HTML markup and C# code that runs on the Web server.
- View has a file extension .cshtml

⇒ 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 C#.NET programming languages.
- The MVC framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.

⇒ Razor Engine:

- Razor engine converts the code of view into HTML.
- Razor engine is used as the default view engine by the MVC framework.
- Compiles a view of your application when the view is requested for the first time.
- Delivers the compiled view for subsequent requests until you make changes to the view.
- Provide template markup syntax to segregate HTML and programming code in a view.
- first requires identifying the server-side code from the markup code to interpret the server-side code embedded inside a view file.
- Uses the @ symbol, to seperate the server-side code from the markup code.

⇒ While creating a Razor View, we should consider following rules:

- Start inline expressions with @
- Enclose code blocks between @ { and }
- Variables are declared with the var keyword
- Enclose strings in quotation marks.
- End a Razor code statement with semicolon(;)

Example:-

Index.cshtml

```
<h1> Welcome </h1>
@ {
    var name = "Pawan";
  }.
<h2> Name = @name</h2>
<h2> Current Year is @DateTime.Now.Year</h2>

@ {
    for (int i=0 ; i<10 ; i++)
    {
      <h2>@i </h2>
    }
    string[] fruits = {"apple", "banana", "mango"};

    foreach(var i in fruits)
    {
      <h2> @ i </h2>
    }
}
```

⇒ Layout View (Master Page)

- Layout view provides consistent look for all the views in a web application.

- Layout View is as same as Asp.net Web forms Masterpage.

- Extension of Layout View is .cshtml
- Default name of LayoutView is _Layout.cshtml

- Layout View files is usually located in Views/Shared.
- An application can have multiple Layout Views.

- Layout Property is used to connect layout View with a view.

* To creat Layout. Go to Solution Explorer → ~~click on~~ Views → Shared → click on shared folder then Add → New item then select Razor Layout.

* To connect the LayoutView file to the child views, insert the following code in the child view.

Layout = "Path of the layoutpage";

Example

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

⇒ View Start file (_ViewStart.cshtml)

- It also work like a master page.
- Using the View Start file we do not need to connect it manually with the child views.

- This voilates the DRY (Don't Repeat Yourself)

- In the _ViewStart file we add our Layout (master page).

Example :

_ViewStart.cshtml

```
@ {
    Layout = "_Layout";
}
```

- It is located into the views folder.
- If we have multiple Layout views then we can call multiple Layout files in our View Start file.

Example :

```
@ {
    if ( True)
    {
        Layout = "_Layout1";
    }
    else
        Layout = "_Layout2";
```

⇒ Passing Data From Controller to View :

- In an ASP.NET CORE 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 following objects to pass data between controller an view:

1. ViewData
2. ViewBag
3. TempData
4. Strongly Typed Views

* ViewData:

- It passes data from a controller to a view.

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 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 you use complex datatype to avoid error.

* Note:- ViewData does not provide compile time error checking. for example - if you mis-spell keys you wouldn't get any compile time errors. You get to know about the error only at runtime.

Example:-

Pass data from controller

UserController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace RoutingWithoutMVC.Controllers
{
  public class UserController : Controller
  {
    public IActionResult Index()
    {
        ViewData["Data1"] = "PawanKumar";
        string[] arr = {"Guitar", "Flute", "Piano"};
        ViewData["Data2"] = arr;
    }  }  }   return View();
```

Access the passed data in view -

Index.cshtml

```
@{
    ViewData["Title"] = "Index";
}

<h2> Welcome @ViewData["Data1"], you
like the instruments; </h2>

@{
    foreach ( var item in (string[])ViewData
                ["Data2"])
    {
        <h2> @item </h2>
    }
}
```

- ViewData will be accesible in only that view in which it is passed.

* **View Bag :**

- It passes data from a controller to a view.

Syntax :

```
ViewBag.<key> = "Value";
```

<Property Name>

PropertyName : is a String value that represents a property of ViewBag.

Value : is the value of the property of ViewBag, Value may be string, object, list, int, array, char, float etc.

ViewData and ViewBag can access each other data interchangeably.

- ViewBag is a dynamic data type 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. and we can access the ViewBag ~~data~~ through the ViewData in the view and vice-versa.

- 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 introduced in C# 4.0.

- ViewBag does not require typecasting when you use complex data type.

*Note: • ViewBag does not provide compile time error checking like mis-spell keys, we will get to know only at runtime.

Example:

Controller

```
public IActionResult Index()
{ ViewBag.Data1 = "Pawankumar";
  ViewBag.Data2 = "Vishwakarma";
}
```

Action / View

```
@ {
    ViewData["Title"] = "Index";
  }
<h2> Welcome @ ViewBag.Data1 </h2>
<h2> Your surname is @ ViewData["Data2"]
</h2>
```

* TempData :

- It passes data from a controller to a view.
- TempData stores data as key-value pairs.
- Syntax:

```
TempData [<key>] = "<Value>";
```

- TempData can be accessed from any of the views unlike the ViewData and ViewBag.

Example:

Controller.cs

```
public IActionResult Index()
{
    ViewData["data1"] = "View Data";
    TempData["data2"] = "Pawankumar";
    return View();
}
```

View

```
@{
    viewData["Title"] = "Index";
}

<h2> Welcome @TempData["data2"]</h2>
```

- TempData is used only for current or subsequent requests as it is a very short-lived instance.
- Redirecting 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 to serve the redirected view.

- Sharing data between ~~the~~ the controller actions are done through the ASP.NET MVC TempData dictionary.

- TempData is a Dictionary object derived from the TempDataDictionary class.

- TempData value must be type cast before use.
- It 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.
- Once TempData is used then it will be cleared.

- Call TempData.Keep() to keep all the values of TempData in a third request.

Example :-

Controller

```
public IActionResult Index()
{
    TempData["Data1"] = "PawanKumar";
    TempData.Keep("Data1");
    return View();
}
public IActionResult About()
{
    TemData.keep("Data1");
    return View();
}
```

Views

Index

```
@{
    viewData["Title"] = "Index";
}
<h2> Welcome @TempData["Data1"]</h2>
```

About

```
@{
    viewData["Title"] = "About";
}
<h2> Welcom @TempData["Data1"] </h2>
```

⇒ Models

- A model is a class with .cs (for C#) as an extension having both properties and methods.

- Models are used to set or get the data.

- If your application does not have data, then there is no need for a model.

- The Models in ASP.NET Core MVC contains a set of classes that are used to represent the domain data (you can also say the business data) as well as it also contains logic to manage the domain/ business data.

- Models in ASP.NET Core MVC Application are used to manage the data.

- We can perform operations like insert, update, delete by using Models.

*Note: It is not Mandatory, but it is a good practice to store all the model inside the Model Folder.

⇒ Repository Pattern:

- With the Repository pattern, we create an abstraction layer between the data access and business logic layer of an application.

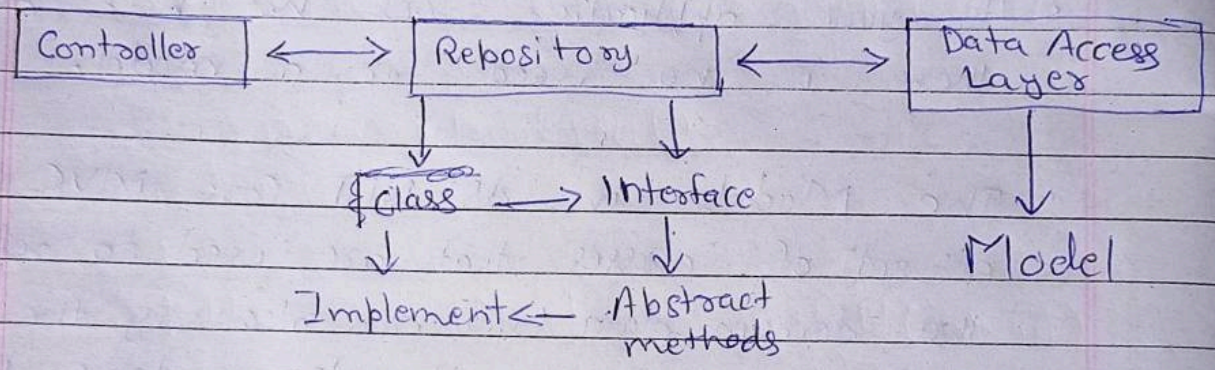


figure 1

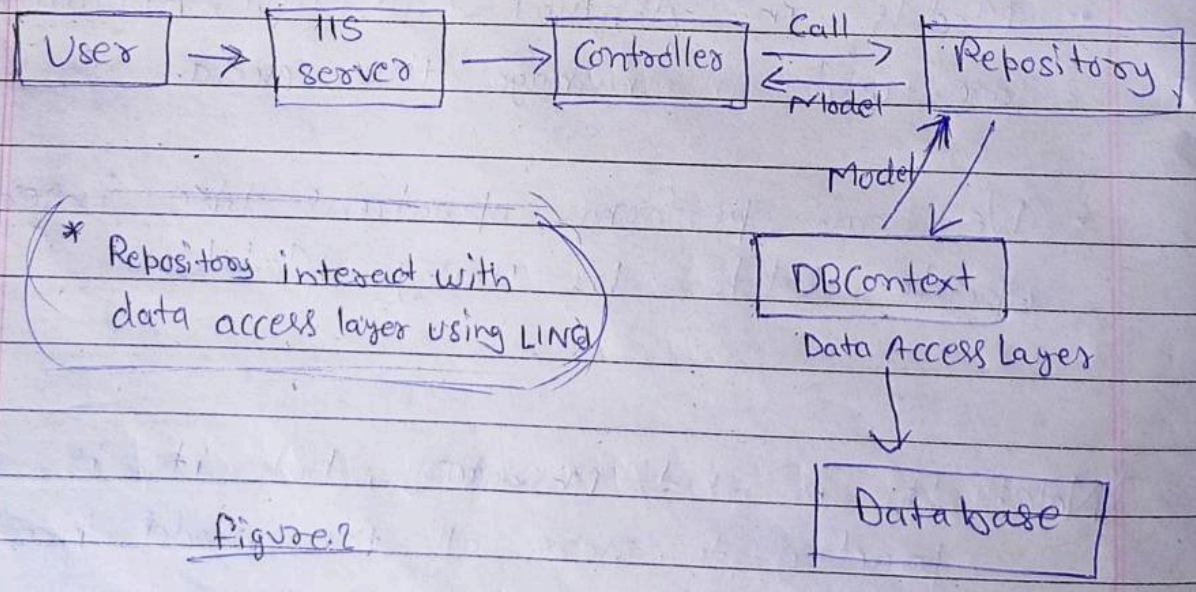


figure 2

- By using it, we are promoting a more loosely coupled approach to access our data from database.
- Also, code is cleaner & easier to maintain and reuse.

Example:-

- First we create a model class to represent a table and its properties.

StudentModel.cs

```
namespace ModelsInAspNetCore.Models
{
    public class StudentModel
    {
        public int id { get; set; }
        public string name { get; set; }
        public int age { get; set; }
        public string gender { get; set; }
    }
}
```

- We will create a New folder in the project named repository.

Then we'll create an interface inside the repository folder to define abstract methods of the repository class.

IStudent.cs

```
public interface IStudent
{
    List <StudentModel> getAllStudents();
    StudentModel getStudentById (int id);
}
```

- Next we'll create a class to define the repository and implement the interface ~~abst~~ methods.

StudentRepository.cs

```
public class StudentRepository : IStudent
{
    public List<StudentModel> getAllStudent()
    {
        return DataSource();
    }

    public StudentModel getStudentById(int id)
    {
        return DataSource().Where(student => student.id
        == id).FirstOrDefault();
            or
    // return DataSource().FirstOrDefault(studen => studen.id == id)
    }

// At this time not using data from database so
creating data manually.

private List<StudentModel> DataSource()
{
    return new List<StudentModel>()
    {
        new StudentModel { id = 0, name = "xyz", age = 23, gender = "M" },
        new StudentModel { id = 1, name = "abc", age = 21, gender = "F" },
        ⋮                ⋮
    };
}
}
```

- Now we'll access the data from repository to our controller.

First implement our repository and then create an object of our repository _studentRepository.

Then create methods GetStudents(), GetStudent to access all the members of the repository.

HomeController.cs

```
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> GetStudents()
    {
        return _studentRepository.getAllStudents();
    }
    public StudentModel GetStudent(int id)
    {
        return _studentRepository.getStudentById(id);
    }
    public IActionResult Index() { return View(); }
}
```

⇒ How to Pass Model Data From Controller to View using ViewData, ViewBag & TempData :

- First Create model and declare the properties.

Ex:

Employee.cs

```
public class Employee
{
    public int EmpId {get; set;}
    public string EmpName {get; set;}

}
```

- Then Access that model and its properties in Controller. (inside any action to pass the data to view)

Ex:

HomeController.cs

```
public IActionResult Index()
{
    List<Employee> employees = new List<Employee>()
    {
        new Employee { EmpId = 101, EmpName = "Pawan" }

    }
    ViewData["Employees"] = employees;
    return View();
}
```

And Now we can access DataView's data in view as earlier

We can do same with ViewBag & TempData also.

* Strongly Typed View

- Strongly typed view 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.

- You can bind any class as a model to view.

- You can access model properties on that view.
- You 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.

- We can access the model data in our views using @model <modelsNamespace.modelname>.

Example :-

Create a model Employee.cs

```
public class Employee
{
    public int EmpId { get ; set ;}
    public string EmpName { get ; set ;}
    public int Salary { get ; set ;}
}
```

HomeController.cs

```
using StronglyTypedViewASPCore.Models;

public IActionResult Index()
{
    Employee emp = new Employee()
    {
        EmpId = 1,
        EmpName = "Pawan",
        Salary = 25000
    };
    return View(emp);
```

Index.cshtml :

```
@model StronglyTypedViewASPCore.Models.Employee
@{
    ViewData["Title"] = "Home Page";
}
@{
    <h2> ID: @Model.EmpId </h2>
    <h2> Name: @Model.EmpName </h2>
    <h2> Salary: @Model.Salary </h2>
}
```

⇒ How to Install Bootstrap, Tailwind, JQuery or other Libraries in ASP.NET Core.

- All static files like bootstrap, CSS, JS, jquery are placed inside wwwroot folder.

- Images folder is also placed in wwwroot folder.

- Steps: Create empty ASPNETCore app and then creat controller.
  1. Create a folder named wwwroot.
  2. Right click on project Name → Add → Client-Side Library.
  3. select cdnjs and search your library, then install

- You can add the libraries in libman.json also.

- You can not access or use libraries without app.UseStaticFiles() method.

- We add app.UseStaticFiles() method in our Program.cs file.

- app.UseStaticFiles(); without this method you can not access static files like bootstrap, CSS, JS, jquery etc.

- Libman.json file contains all the configurations for clients side libraries.

⇒ View Imports file in ASP.NET Core (_ViewImports.cshtml)

- In ASP.NET Core MVC Application, the _ViewImports.cshtml file provides a mechanism to include the directive globally for Razor Pages / Views so that we don't have to add them individually in each and every page.

- ~~ViewPorts.~~
- ViewImports.cshtml file supports the following directives:
  - @addTagHelper
  - @tagHelperPrefix
  - @removeTagHelper
  - @namespace
  - @inject
  - @model
  - @using

- _ViewImports file is hierarchical order

```
* Views
    * Home
        . [_ViewImports.cshtml]
        . Index.cshtml
  _ViewImports.cshtml
```

- If _ViewImports.cshtml present in Home then Index will use that file otherwise Index will use _ViewImports file inside of Views folder

⇒ Tag Helpers

- Tag Helpers are basically special attributes provided by Asp.net Core.

- Tag Helpers enable server-side components to participate in creating and rendering HTML elements in Views.

- Tag helpers are a new feature and similar to 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 attributes are applied.

*Note: Before start working with tag helpers, make sure you have included namespace for tag helpers in your ViewImports file.
ie. Microsoft.AspNetCore.Mvc.TagHelpers.

- Add this line in your ViewImports file.

  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

- @addTagHelper is a directive.

Syntax:

```
<a asp-controller="<controllername>" asp-action="<viewname
> Text </a>
```

Example:-

index.cshtml

```
@{
    ViewData["Title"] = "Home Page";
}

<a href="/Home/Contact"> Contact Page 1 </a> -> Normal HTML
<br/>
@Html.ActionLink("Contact Page 2", "Contact", "Home")
<br/>          => This is used in MVC 5
<a asp-controller="Home" asp-action="Contact"> Contact Page 3 </a>
```

Image Tag Helper

- In order to use the image tag Helper in ASP.NET Core Application, you need to add the asp-append-version attribute to the <img> tag and need to set the value to true.

- Image tag helper provides cache busting behavior within asp.net core application.

- In general, when we visit a web page & if that web page contains some image, then most of the modern web browser cache the images for later use.

- In future, when we revisit to that page then ~~that~~ the browser loads the images from ~~the~~ cache instead of downloading the images from the server.

Example:

```
<img src="~/images/ASPNETCORE.jpeg" asp-append-version="true"
height="200" width="400" alt="ASPNETCORE" />
```

⇒ Creating Form using Tag Helpers

- The asp-for attribute is arguably one of the most common tag helpers that you 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.

- Eg: asp-for = "Movie.Year" when retrieving data from a POST request.

- asp-for sets the id, name and validation related attributes.

There are many tag helpers to create forms

- The Form Tag Helper
- Form Action Tag Helper
- Input Tag Helper
- Textarea Tag Helper
- Label Tag Helper
- Validation Tag Helpers.
- Select Tag Helper

Example :

- Create a model Employee.cs

```
namespace TagHelpers.Models
{   public class Employee
    {   public string Name { get; set; }
        public int Age { get; set; }
        public string Designation { get; set; }
        public int Salary { get; set; }
    } }
```

- Create form in Index View (Index.cshtml)

```
@model Employee;          --> Bind the Employee model
@{ ViewData["Title"] = "Home Page"; }

<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" class="form-control" /> <br/>
    <input asp-for="Age" class="form-control"/> <br/>
    <input asp-for="Designation" class="form-control"/><br/>
    <input asp-for="Salary" class="form-control"/><br/>
    <input type="submit" value="Submit" class="btn
      btn-outline-primary btn-block"/>
    </form>
    </div>
  </div>
</div>
```

⇒ 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, floats or they may be complex types like Student, Employee, Customer, Order etc.

Example with simple data types

HomeController.cs

```
public string Test(int id, string name)
{
    return "Id is: " + id + " Name is: " + name;
}
```

O/P (i) https// - - - /Home/Test/5 ? name = Pawan
RouteData
Query string
→ Id is 5 Name is Pawan

(ii) - - - /Home/Test/5 ? name = Pawan & id = 7
→ Id is 5 Name is Pawan

(iii) - - - - /Home/Test/ ? name = Pawan & id = 7
→ Id is 7 Name is Pawan

- When MVC recieves 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 method's 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:

1. Form Values: Values that go in HTTP request using POST method.

2. Route Values: The set of route values provided by routing.

3. Query String: The query string part of the URL
ex. /?id=1 & name=Pawan

* Note: Form values, route data, and query strings are all stored as name-value pairs.

⇒ Model Validation

(Often the data entered by the user is not valid and can not be saved into the database)

- The entered data may contain a typo or user may intentionally enter the inappropriate data.
- Hence, we need to validate the user input before storing it in the database.

* ASP.NET Core gives us Model Validator, which uses the validation attributes to validate the model, which makes our task easier.

- The form data is posted to Controller action is automatically mapped to the action parameter by the Model Binder.

- 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 recieved from the client.

* Explicitly Validating a Model

- Once you received the model in the controller, you can validate the model programatically as shown:-

```
if ( string.IsNullOrEmpty(model.Name))
{
    //validation failed
    // send the list of errors to client.
}
```

- 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 Model Validator runs after the model binding and runs a series of validations on each property of the model based on the attributes that you set on the model property.

→ These are some Built-in attributes

- [Required]
- [Regular Expression]
- [String Length]
- [Compare]
- [Email Address]
- [Phone]
- [Range]
- [Url]
- [Min Length]
- [Max Length]

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

⇒ Validation Tag Helpers :

- There are two validation tag helpers in asp.net core.

(i) Validation Message Tag Helper :

It displays the validation message for a single property of a Model. Eg : asp-validation-for

(ii) Validation Summary Tag Helper :

It displays a summary of validation messages for all the properties of a Model.

Eg: asp-validation-summary

Ex :

(i)
```
<form asp-controller="Home" asp-action="Index" method="post">
<input asp-for="Name" class="form-control">
<span asp-validation-for="Name"></span>
</form>
```

(ii)
```
<div asp-validation-summary="All" style="color: red;">
</div>
<form ... >
<input asp-for="Name" class="form-control">
</form>
```

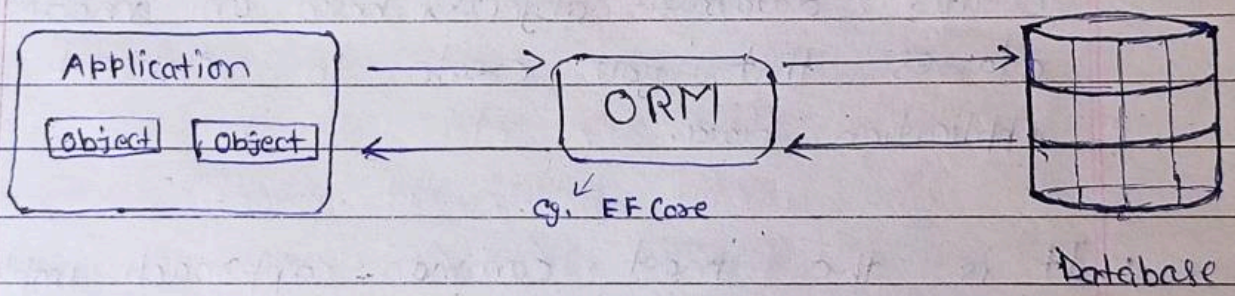
⇒ Entity Framework Core

- Entity framework core is an ORM framework.

* **ORM:**
  To address the data access requirements of ASP.NET CORE MVC application, you 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.



- Entity Framework Core is the new version of Entity Framework after EF6 but it is redesigned.

- It is open-source, lightweight, extensible and a cross-platform version of Entity Framework.

- EF core is a data access technology.

- EF Core is intended to be used with .NET Core applications.
  It can also be used with standard .NET 4.5 + framework based applications.

* EF Core supports two developement approaches.

  1) Code-First
  2) Database-First

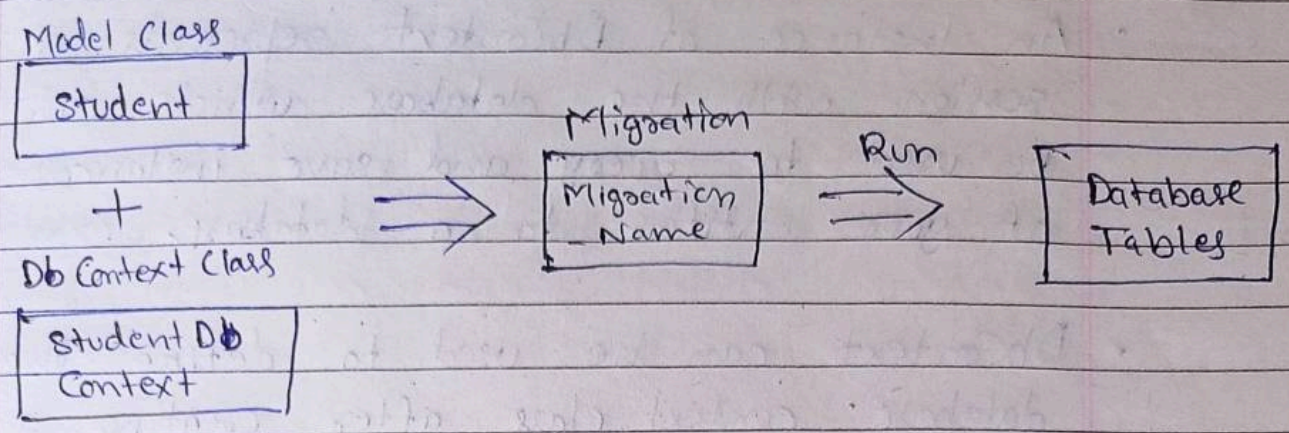
* Code-First Approach

- In the code-first approach the EF Core creates database objects (tables) based on model classes that you create to represent application data.

- It is the most common approach implemented in ASP.NET Core MVC framework.

- It allows us to develop our application by coding model classes and properties and delegate the process of creating the database objects to the EF Core.

- The code-first approach allows us to define our own model by creating custom classes.



Dig. Code-first approach

* Steps to use Code-first Approach :

Step 1: Install 3 packages in your ASP.NET Core MVC app.
(i) Microsoft.EntityFrameworkCore.SqlServer
(ii) Microsoft.EntityFrameworkCore.Tools
(iii) Microsoft.EntityFrameworkCore.Design

Step 2: · Create a Model Class.
· Create a DbContext class.

· DbContext class is an integral part of EF.

· We use it in our application code to interact with the underlying database.

· DbContext class 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 your entities to a database.

- DbContext can be used to define the database context class after creating a model class.

- It uses 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:

- 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.

Step 3: · Create a Connection String in appsettings.json file.

Step 4: · Registering Connection string in Program.cs file.

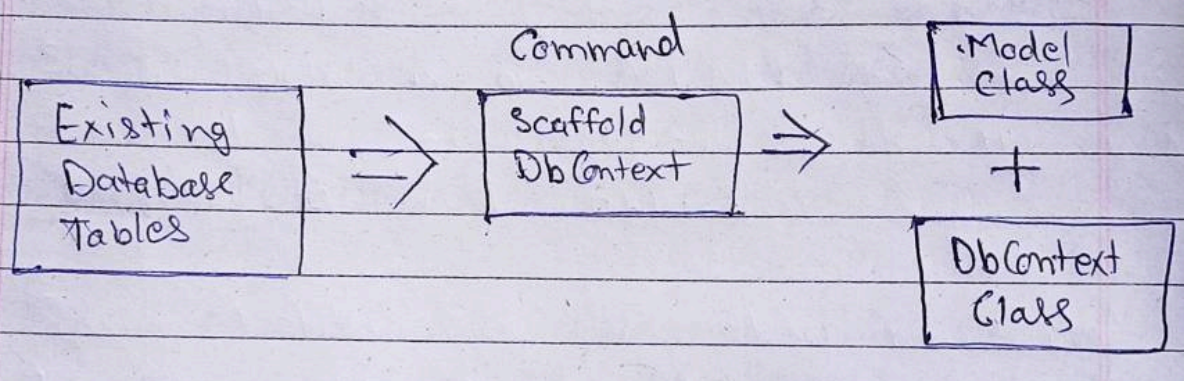
Step 5: Add a migration and run the migration.

* Note: We use [ValidateAntiForgeryToken] ~~befor~~ post method ~~avoid~~ to avoid CSRF (Cross Site Request Forgery

* Database-First Approach

- In the database-first approach the Entity Framework core creates model classes and properties corresponding to the existing database objects, such as tables and columns.

- The database-first approach is applicable in scenario where a database already exists for the application.



Dig. DB-first Approach

* Steps to use DB-first approach:-

Step 1: Install 3 Packages in your ASP.NET Core MVC application.

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

Step 2: 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 the how we can update our Model and DbContext :-

By using the following command :-

```
Scaffold-DbContext "server= ServerName; database=
DatabaseName; trusted_connection = true"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir
Models -force
```

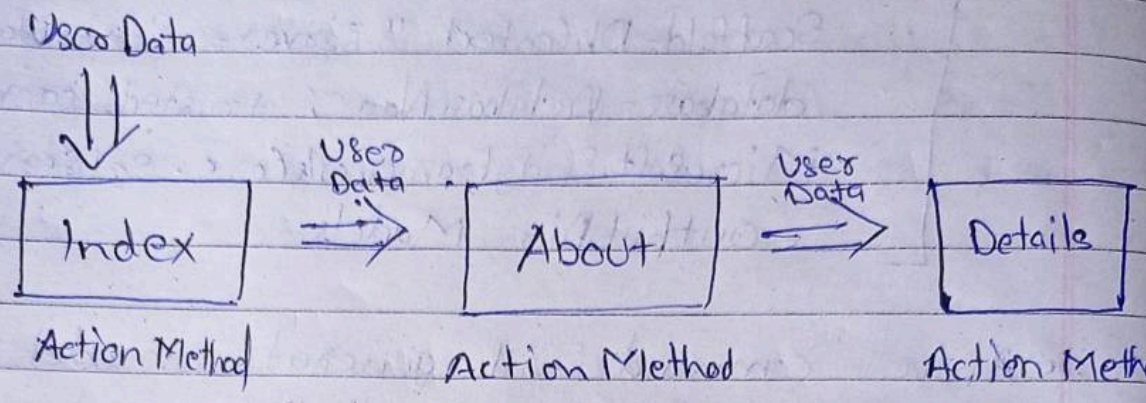
Step 3: Move Connection String from DbContext Class to appsetting.json file.

Step 4: Registering Connection String in Program.cs file.

write the following lines after builder.Services.AddControll...

- var provider = builder.Services.BuildServiceProvider();
- var config = provider.GetRequiredService<IConfiguration>();
- builder.Services.AddDbContext<CodeFirstDBContext> (item =>
  item.UseSqlServer(config.GetConnectionString("dbcs")));

⇒ Session



- It is a state management technique.
- Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app.
- Session state uses a store maintained by the app to persist data across requests from a client.
- 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 are not shared across browsers.

- Session 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.

- Session persist 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.

- When many users access an application simultaneously then each of these users will have a different session state.

- Every session has a unique session id.

* Steps to create and use the session:-

Step 1: · Add below line before build. in Program.
" builder.Services.AddSession();

· Add below line after build ~~and~~ before app.run().

" app.UseSession();"

Step 2: · Create Session Variable in our Controller.
HttpContext.Session.SetString("MyKey", "Value");

We have to ~~write~~ " using Microsoft.AspNetCore.Http;
for accessing the session.

· We can set different type of values in a session.

Step 3: · Access Session variable in ~~views~~ in controller

"HttpContext.Session.GetString("MyKey");

To access the session in view we write these lines in the view.

@using Microsoft.AspNetCore.Http
@inject Microsoft.AspNetCore.Http.IHttpContextAccessor accessor

<h1> @accessor.HttpContext.Session.GetString("MyKey") </h1>

⇒ Creating DropDownList in Asp.Net Core 6 (using SelectList | Tag Helper)

- **Method 1**

HomeController.cs

```
public enum Gender
{
    Male, Female
}
```

Index.cshtml

```
@using <project_name>.Controllers;
<label class="form-label">Select Gender:</label>
@Html.DropDownList("Gender", Html.GetEnumSelectList
<Gender>(), "Select Gender", new {@class="form-control"})
```

- **Method 2**

HomeController.cs

```
public IActionResult Index()
{
    List<SelectListItem> Gender = new()
    {
        new SelectListItem { Value="Male", Text="Male"},
        new SelectListItem { Value="Female", Text="Female"},
    };
ViewBag.Gender = Gender;
return View();  }
```

Index.cshtml

```
@Html.DropDownList("Gender", ViewBag.Gender,
"Select Gender", new {@class = "form-control"})
```

or

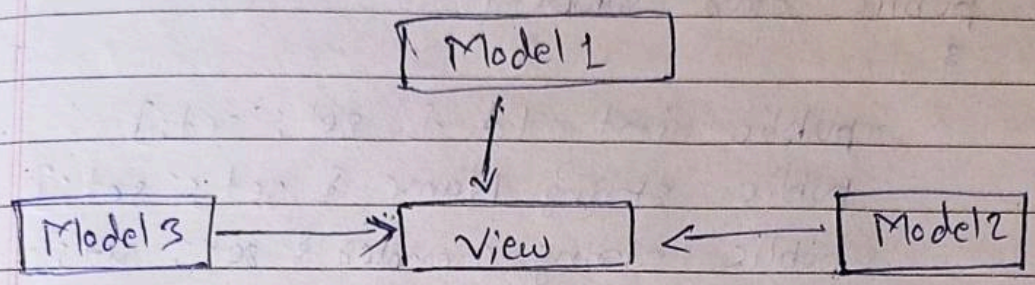
```
<select name="Gender" asp-items="@ViewBag.Gender"
    <option value=" "> Select Gender </option>
</select>
```

class = " form-control

* Binding DropDownList with Database

Practical Implementation

⇒ Bind Multiple Models with a Single View



Example:- We have 2 models Teacher & Student if we want to show both models data in our single view then we have to bind both models with our view.

But ASP.NET Core doesn't allow to bind multiple models with a single view.

To achieve this we create (viewModel) another model class and bind our models in that and then we bind that viewModel to our view.

Ex. we have Two models Teacher.cs and student.cs and we will create another viewModel class CollegeViewModel.cs

Teacher.cs

```
public class Teacher
{   public int Id {get; set;}
    public string Name {get; set;}
    public string Qualification {get; set;}
    public int Salary {get; set;}
}
```

Student.cs

```
public class student
{
    public int Id { get; set;}
    public string Name { get; set;}
    public string Gender { get; set;}.
    public int Standard { get; set;}
}
```

CollegeViewModel.cs

```
public class CollegeViewModel
{
    public List<Student> MyStudents { get; set;}
    public List<Teacher> MyTeachers { get; set;}
}
```

HomeController.cs

```
public IActionResult Index()
{
    List<Student> students = new List<Student>
    {
        new Student { Id=1, Name="Pawan", Gender="M", Std=12},
        :       :       :       :
    }
    List<Teacher> teachers = new List<Teacher>
    {
        new Teacher { Id=1, Name="Dinesh", Qualification="MCA", Salary=20
        :       :       :       :
    }
```

```
CollegeViewModel cvm = new CollegeViewModel()
{
    MyStudents = students,
    MyTeachers = teachers
};
return View(cvm);
}
```

- Now we can use the cvm ~~model~~ data in our Index view.

Index.cshtml

```
@ ViewModels.Models.CollegeViewModel

<div class="container">
  @ if (Model.MyStudents.Any())
  {
      <table>
        <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>
              :  :  :  
```

⇒ # Partial View (_partialView.cshtml)

- 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 single partial views in different multiple views.

- Partial views can not be used seperately, we have to attach in some other view.

- We create the partial view in shared folder under the views folder.

- To create partial view
  - Right Click on Shared folder
  - Then click on Add → View → Razor View
  - * Please remember to click on the checkbox of the partial view when creating the ~~View to Create~~ partial view. ☐ Create as a partial view

- Syntax to use partial view in the main view.

```
<partial name="<partialViewname>" />
```

Ex. <partial name="_EmployeeNames" />

- We can use partial view multiple times in single v

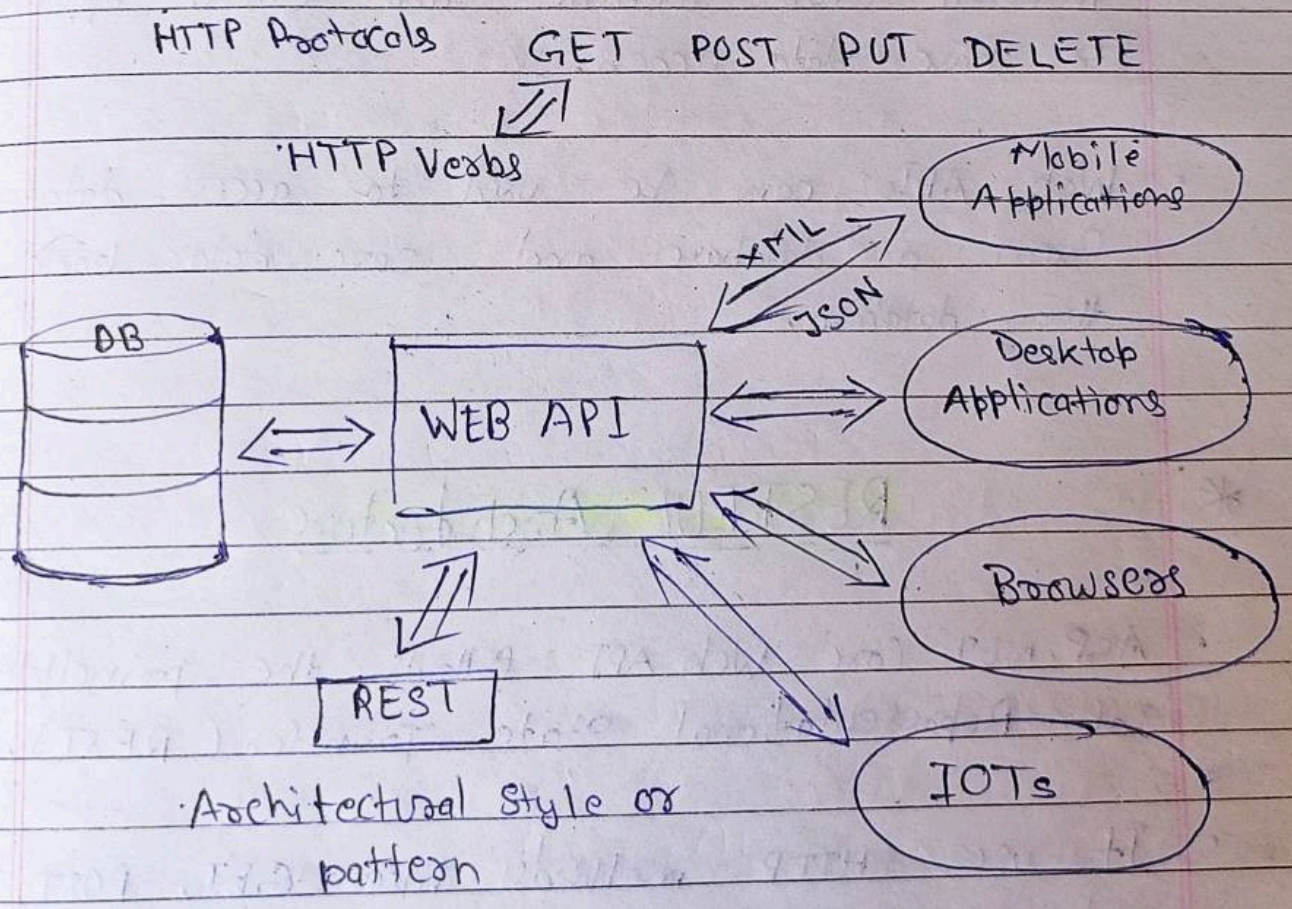
Web API

→ ASP.NET Core Web API (Restful APIs)

- Before we understand what is Web API, Let's see what is an API (Application Programing Interface).

- API is a set of subroutine definitions, protocols, and tools for building software & applications.

- Web API is basically a url which returns data.



- Web API is used to enable communication or interaction with software components with each other.

- A web API is a set of endpoints that allow different software applications to communicate with each other over the internet.

- ASP.NET Core Web API is a framework that allows us to build and create RESTFul APIs using ASP.NET Core platform.

- Web API is often used to provide an interface for websites and client applications to have data access.

- Web APIs 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 resources identified by URLs.

- In REST architecture, a REST server simply provides access to resources and the REST client accesses and presents the resources.

- REST uses various representations to represent a resource like Text, JSON and XML, mostly JSON.
- A Restful system consists of:
  - client ⇒ who requests for the resources.
  - Server ⇒ who has the resources.

- REST architectural 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

* Swagger:

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

* How to Create an API :

Step 1: Create new project ASP.NET Core Web API template.

Step 2: Give name and click next and then click the checkboxes ☐ Use controllers
☑ Enable OpenAPI support
Then click on create.

Step 3: Define the API endpoints in the created project.
- Open controllers folder and find default "WeatherforecastController.cs".
- You can modify it or create new.

- How to create a new ~~Contro~~ API controller.

- Right click on controllers folder and click on add → Controller
Then select the "API controller Empty" under the ~~API~~ Common → API. instead of MVC.

Example:

FruitsAPIController.cs

```
namespace ASPCoreWebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class FruitsAPIController : ControllerBase
    {
        public List<string> Fruits = new List<string>()
        {
            "Apple", "Banana", "Grapes", "Mango"
        };

        [HttpGet]
        public List<string> GetFruits()
        {
            return Fruits;
        }

        [HttpGet("{id}")]
        public string GetFruitsById(int id)
        {
            return Fruits.ElementAt(id);
        }
    }
}
```

⇒ # Authentication and Authorization

- **Authentication** is process of checking the identity of the user that accessing our application.

- **Authorization** is process of validating privileges / authorities to access a resource of the application.

⇒ Identity Framework

- In ASP.NET Core Identity is a framework for managing and storing user accounts in ASP.NET Core applications.

- 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 Gmail 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.

- Identity framework manage users, password, profile data, roles, tokens, email confirmation.

THANK YOU!