**Dot net Core MVC Notes with Project**

* **Chapter 2**

**Introduction to HTTP**

HTTP stands for hypertext transfer protocol which contains a certain set of rules for communication between browser and server. HTTP sends requests from browser to server and gets responses in return from server to browser.

**Response Status Code**

| **101** | Switching Protocol | Ex switching from http to https |
| --- | --- | --- |
| **200** | OK | Request is successful |
| **302** | Found | Indicates redirection |
| **304** | Not Modified | Indicates some files are loaded from cache memory |
| **400** | Bad Request | Indicates some required data sent to request is incorrect or missing. |
| **401** | Unauthorised | Credentials are incorrect |
| **404** | Not Found | URL does not exists |
| **500** | Internal Server Error | Any runtime error in code represents 500 |

**HTTP Request**

When we send or seek information from browser to server it is called HTTP Request.

**Query String**

Query string is syntax where you can send parameters along with the url. Part after the question mark in the below example is the query string.

Ex /dashboard?id=1

**HTTP Request Types**

| **GET** | Retrieve information |
| --- | --- |
| **POST** | Insert information into database |
| **PUT** | Update information |
| **PATCH** | Partially update information |
| **DELETE** | Used to delete resource from databse |

**Chapter 3**

**Middleware**

Middleware is code logic that is injected into the application pipeline to handle request and response. Each request goes through this pipeline and middleware decides whether to pass the request to the next middleware or not. Middleware are chained one after another and executed in the same sequence as they are added. Middleare can be a request delegate(anonymous method or lambda expression) or a class. You can define middleware in the program.cs file. Once you call builder.Build() method you will get an application builder object. This object is used to create middleware.

**app.Run()**

The extension method called Run is used to execute a terminating/short-circuiting middleware that does not forward requests to the next middleware.

Ex

app.Run(async (HttpContext context) =>

{

await context.Response.WriteAsync("Hello world");

});

**app.Use()**

app.Use() extension method we can pass requests to the next middleware as well as it is also able to short circuit requests.

Ex

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

{

//Some logic

await next(context);

});

**Custom Middleware**

Custom Middleware is a class that contains some complex logic and used to separate middleware logic from lambda expression (request delegates) to reusable class. The Middleware class needs to implement an interface called IMiddleware which contains the InvokeAsync method.

Class MiddlewareClassName : IMiddleware

{

Public async Task InvokeAsync (HttpContext context, RequestDelegate nex)

{

//some logic

Await next(context);

}

}

**How to add Custom Middleware**

| Create Class | First step is to create a class first that contains required logic that should be inherited by IMiddleware interface like below example.  namespace MyFirstApp.CustomMiddleware  {  public class MyCustomMiddleware : IMiddleware  {  public async Task InvokeAsync(HttpContext context, RequestDelegate next)  {  await context.Response.WriteAsync("my custom middleware - start");  await next(context);  await context.Response.WriteAsync("my custom middleware - ends");  }  }  } |
| --- | --- |
| Register Service | Register newly created middleware class.  builder.Services.AddTransient<MyCustomMiddleware>(); |
| Use middleware | app.UseMiddleware<MyCustomMiddleware>(); |
| Use of Extension method | Also you can use extension method which is used to invoke middleware with single method call  public static class CustomMiddlewareExtension  {  public static IApplicationBuilder UseMyCustomMiddleware  (this IApplicationBuilder app)  {  return app.UseMiddleware<MyCustomMiddleware>();    }  }  app.UseMyCustomMiddleware(); |

**Right Order of Middleware**

1. ExceptionHandler
2. HSTS(Http strict transport security)
3. HttpsRediretion
4. Static Files
5. Routing
6. CORS
7. Athentication
8. Authorization
9. Custom Middlewares
10. EndPoint

**Middleware UseWhen**

Like use extension method we have UseWhen which check if certain condition is true then it will process with some other middleware branch.

#### **Questions for this assignment**

1. What is middleware?

Ans: Middleware is custom logic that you want to implement and form as a pipeline so that each request and response will go through this code logic. Middleware further responsible for validating this request then decide whether to pass it for next middleware or terminate (short circuit) it.

1. What is the difference between IApplicationBuilder.Use() and IApplicationBuilder.Run()?

Ans: app.Run() is extension method for IApplicationBuilder object which is responsible for short circuiting current requests. It accepts HttpContext objects as parameters in lambda expression.

app.Use() is an extension method which accepts httpcontext and request delegate object as next in method and responsible to validate current request and also we can have logic to pass request to next middleware for further processing.

1. What is the use of the "Map" extension while adding middleware to the ASP.NET Core pipeline?

Ans : It is used for branching the pipeline if request starts with specified path respective branch of pipeline get executed.

1. How do you create a custom middleware?

Ans :

There are two ways to create custom middleware

| Step 1 Create a Middleware class that inherits the IMiddleware interface. |
| --- |
| Step 2 Implement InvokeAsync method from IMiddleware interface and write logic as per your requirement. |
| Step 3 Register middleware class before builder.Build() method using required lifecycle scope as per your requirement |
| Step 4 define newly created middleware in middleware pipeline either using below two approaches  app.UseMiddlware<YourCustomMiddlewareName>();  Or you can create static method in middleware class which accepts application builder object then you can create extension method  app.UseYouCustomMiddlewareName(); |

Second Approach

| Step 1 click on add new Item select middleware class |
| --- |
| Step 2 this will give you the default template for middleware class where request delegate is injected through constructor, it has invoke async method where you need to write your logic and also it provides method extension for application builder object. |
| Step 3 again you need to register this middleware and define it in the middleware pipeline using the extension method.  app.UseYouCustomMiddlewareName(); |

1. What is the right order of middleware used in production-level applications?

Ans:

| Exception Handling |
| --- |
| HSTS (Http Strict Transport Security) |
| HTTP Redirection |
| Use Static files |
| Use Routing |
| Use CORS |
| Authentication |
| Authorization |
| Custom Middleware |
| Endpoint |

**Chapter 3**

**Introduction to Routing**

Routing is functionality of mapping in coming requests with Routing Handler so specific action methods get invoked. To use routing we need to use **app.UseRouting()** middleware and **app.UseEnpoints()** are used to create routing mapping.

**GetEndPoint()**

This method is used to get instances of end points that are defined in the system. We need to make sure that before this we should use app.UseRouting() else this will return null value.

**Route Parameters**

URL consists of some fixed part and some dynamic part is called lateral and dynamic part is called Route Parameter. In below example

file/sample.txt

File is lateral part and sample and txt is route parameter we can define it like below

file/{filename}.{extension}

**Optional Parameter**

In routing to make parameter as optional we need to assign null as default parameter. To make routing parameter as optional parameter we need to indicate it with ? like below

“{parameter?}”

Ex

"files/{filename?}.{extension?}"

**Route Constraint**

If you want to restrict parameters then we can use constraints. Constraints have been stored as part of variables. It makes sure that parameters should be in a specific data type. Few data types are allowed but you can not use all the datatype as route constraint.

Ex

"employee/profile/{employeename=string?}"

Allowed data types

Int, datetime, guid, decimal, long,string

**End Point Selectoin Order**

So for example we have one url that are routing to two different function which one is getting executed so considering this scenario it gives preference based on below 4 points

| URL with more segments  Ex “a/b/c/d” is higher than “a/b/c” |
| --- |
| URL template with literal text has more preference  Ex “a/b” is higher than “a/b/c” |
| URL template that has parameter segment with constraints has more preference  Ex”a/b:int” is higher than “a/b” |
| Catch all parameters(\*\*)  Ex “a/b” is higher than “a/\*\*” |

**WebRoot and UseStaticFiles**

Applications may contain static files generally working with UI. We require all these files but by default dot net core does not serve static files we need to enable it using UseStaticFiles() middleware. We need to store these static files in to webroot folder and default name for it is wwwroot. If you want to add a folder with another name you need to configure it in the web builder method.

**Chapter 4**

**Introduction to Controllers**

Controller is a class that is used to group up a set of actions. Controller is collection of action methods.

**How to create Controller**

| Add controller class |
| --- |
| Register controller class as service using builder.Service.AddController() |
| Enable routing middleware app.UserRouting() |
| Define endpoints middleware  app.UseEndpoints(endpoints =>{  endpoints.MapController();  }) |
| Create public class in controller define your methods define routing and thats it  public class HomeController  {  [Route("home")]  public string method1()  {  return "Hello world";  }  } |

**Return Types**

| **ContentResult :** Content result represent any type specified my MIME type of result like plain text, html data etc  Ex  public ContentResult Index()  {  //return Content("Hello World", "text/plain");  return Content("<h1>Hello World</h1>", "text/html");  } |
| --- |
| **JsonResult :** Represent data in javascript object notation format.  Ex {“firstName”:”John”, “lastName”:”Abraham”}  public JsonResult Person()  {  Person p = new Person()  {  Id = 1,  FirstName = "Durgaprasad",  LastName = "Mohite",  Age = 30  };  return new JsonResult(p);  } |
| **FileResult :** File result sends the content of file as response.  Eg:pdf, xls, txt, exe file etc.  There are three types of file result  Virtual File Result , Physical File Result and File content result   1. Virtual File Result : When your file is saved in root folder then you can use this option   [Route("file-download")]  public VirtualFileResult fileDownLoad()  {  return new VirtualFileResult("/sample.pdf", "application/pdf");  }   1. Physical File Result : When your file is saved in another folder than root folder then you can use this option   [Route("file-physical-download")]  public PhysicalFileResult filePhyscialDownLoad()  {  return new PhysicalFileResult(@"C:\Users\DURGA\Desktop\Study\aspnetcore\ControllerExample\ControllerExample\assignment Dot Net.pdf", "application/pdf");  }   1. File Content Result : In a real world scenario you may have files saved in a database in byte format and you want to return them.then you can use this.   [Route("filedownloadcontent")]  public FileContentResult fileContentDownLoad()  {  byte[] bytes = System.IO.File.ReadAllBytes(@"C:\Users\DURGA\Desktop\Study\aspnetcore\ControllerExample\ControllerExample\assignment Dot Net.pdf");    return new FileContentResult(bytes, "application/pdf");  }    **IActionResult :** In asp.net core it is recommended that all action methods need to be returned as IActionResult because it is the parent interface of all result classes. So by mentioning ActionResult you actually have any type of result which is a subtype of IActionResult. |
| **Status Code :** We can also return a status code based on our requirement. If you want to give content with status code then it is best to use. Below are some types of status code  BadRequest - Represent invalid values passed (400)  NotFountResult - Resource not found (404)  StatusCodeResult - Use this when you want to pass specific status code as result  UnauthorizedResult - Represent response with unauthorised |
| **Redirect Result:** Redirect result sends either HTTP 302 or 301 response to the browser in order to redirect to specific action or URL   | **301** | **302** | | --- | --- | | Permanent redirection  RedirectToActionPermenent() | Temporary redirection  RedirectToAction(); |   **RedirectToActionResult :** Used to return on other URL  LocalRedirectResult - used to redirect to url used in same application  RedirectResult - used to redirect to any other url |

**Chapter 5**

**Overview of Model Binding**

Model Binding is a feature of asp.net core that reads values from http request and passes them as arguments to the action method. Data can be sent using below ways

1. Form Fields : Basically every application consists of forms fields in which forms are submitted as parameters using model binding. Generally values are added in the request body added in two format

* form-urlencoded (default) - instead of sending request details in query parameter you add it in request body
* Form-data -

1. Request body
2. Route Data
3. Query string Parameter

Top one has higher priority and the last one has lower priority.

**Models**

Model is a class that represents structure of data as properties that you would like to receive from the request and send the response. Also known as POCO (Plain Old CLR Objects)

**Model Validations**

Validations are used to make sure data must satisfy certain conditions. In the traditional approach we need to write all the validation code for all objects but now we can do it using attributes. By using this we are able to add validation rule above properties.

**ModelState**

Model state is property of controller base class that is available for all action methods in the controller which is used to check status of validation. After the model binding model validation stage occurs which checks all validation and if there are any validation errors then it will be stored in model state. It contains three properties mainly IsValid, Values and ErrorCount

IsValid - Specify if there is at least one validation error or not.

Values - Contains each model property with corresponding error.

ErrorCount - Returns number of errors.

Ex

public IActionResult Index(Person p)

{

if(!ModelState.IsValid)

{

List<string> errorsList =

//new List<string>();

ModelState.Values.SelectMany(value =>

value.Errors).Select(err =>

err.ErrorMessage).ToList() ;

string errors = string.Join("\n", errorsList);

return BadRequest(errors);

}

return Content($"{p}");

}

**Model Validation**

* **[Required(ErrorMessage=”value”)]**

Specifies that property value is required cant taek blank or empty

* **[StringLength(int maximumLength, MinimumLength=value, ErrorMessage=”Value”)]**

Specifies min and max length allowed in string

* **[Range(int minimum, int maximum, ErrorMessage=”value”)]**

Specifies min and max allowed for numerical values

* **[RegularExpression(string pattern, ErrorMessage=”value”)]**

Specifies the valid pattern

* **[EmailAddress(ErrorMessage=”value”)]**

Specifies that the value should be a valid email address.

* **[Phone(ErrorMessage=”values”)]**

Specifies that the value should be valid phone number

* **[Compare("Password", ErrorMessage ="Password and Confirm password is not matching")]**

For comparing one property with other

* **CustomValidator**

You can also add validation logic as per your requirement

**Bind and BindNever**

* **[Bind] -** Bind attribute specifies that only specified properties should be included in the model binding. Benefit of this is we can avoid over posting of nor required properties.
* **[BindNever] -** Bind never attribute specifies that the specified property should not be included in model binding.

**FromBody**

It enables the input formatters to read data from request body as JSON or XML only

Public IActionResult ActionMethodName([FromBody] type parameter)

{

}

To enable Input formatter we need to register below server in startup class or program class

builder.Services.AddControllers().AddXmlSerializerFormatters();

**Chapter 6**

**Razor Views**

**MVC**

Model View Controller is an architectural pattern that separates application code into three main categories Models, Views and Controllers. Any request comes that actually first reaches the controller action method and then the controller invokes the respective business model to perform required operation. Once operation is completed the result is sent back to controller and then controller further invokes view for displaying result.

**Benefits of MVC**

* Separation of concern
* Each components (MVC) developed independently
* Debugging is easy
* Unit testing each individual component is easier

**View**

View is web page (.cshtml) that is responsible for containing presentation logic that merges data along with static design code (HTML)

**Razor code block**

Razor code block is a C# code block that contains one or more lines of C# code that contain any statements and local functions

**Razor Expression**

@Expression or @(Expression) Razor expression is a C# expression accessing field property or method that returns a value.

Example

| @using ViewsExample.Model;  @{  string appTitle = "Asp.Net Core Demo App";  Person p = new Person() { Name = "John", DOB = Convert.ToDateTime("1989-11-24") };  }  <!DOCTYPE html>  <html>  <head>  <title>@appTitle</title>  <meta charset="utf-8"></meta>  </head>  <body>  <h1>@appTitle</h1>  <h2>Hello @p.Name</h2>  @{  double a = Math.Round( (DateTime.Now - p.DOB).TotalDays / 365);  }  <h3>and I am @a years old</h3>  </body>  </html> |
| --- |

**Razor If**

When you want to show some content based on some condition then you can use Razor if or Razo if else

Syntax

**@if(condition){ @if(condition){**

**c#/html code c#/html code**

**} }**

**else**

**{**

**c#/html code**

**}**

Example

| @using ViewsExample.Model;  @{  string appTitle = "Asp.Net Core Demo App";  Person p = new Person() {  Name = "John",  DOB = null };  }  <!DOCTYPE html>  <html>  <head>  <title>@appTitle</title>  <meta charset="utf-8"></meta>  </head>  <body>  <h1>@appTitle</h1>  <h2>Hello @p.Name</h2>  @if (p.DOB.HasValue)  {  double a = Math.Round( (DateTime.Now - p.DOB).Value.TotalDays / 365);  <h3>my age is @a</h3>  }  else  {  <h3>Date of birth not provided</h3>  }  </body>  </html> |
| --- |

**Razor Switch**

It is same like switch case in C# when ever any case matches with condition we can display some html content

**Syntax**

**@switch(variable){**

**case value1: C#/HTML code; break;**

**case value2: C#/HTML code; break;**

**default : C#/HTML code; break;**

**}**

Example

| @using ViewsExample.Model;  @{  string appTitle = "Asp.Net Core Demo App";  Person p = new Person() {  Name = "John",  DOB = null,  Gender = Gender.male  };  }  <!DOCTYPE html>  <html>  <head>  <title>@appTitle</title>  <meta charset="utf-8"></meta>  </head>  <body>  <h1>@appTitle</h1>  <h2>Hello @p.Name</h2>  @if (p.DOB.HasValue)  {  double a = Math.Round( (DateTime.Now - p.DOB).Value.TotalDays / 365);  <h3>my age is @a</h3>  }  else  {  <h3>Date of birth not provided</h3>  }  @switch(p.Gender)  {  case Gender.male:  <p>International Mens day Nov 19</p>  ;  break;  case Gender.female:  <p>International Womens day Nov 19</p>  ;  break;  case Gender.other:  <p>International Transgender day</p>  ;  break;  }  </body>  </html> |
| --- |

**Razor Foreach**

Razor foreach is used to iterate through collection of and print or display element in form of HTML code

Syntax

**@foreach(var variable in collection)**

**{**

**C#/HTML code here**

**}**

| @using ViewsExample.Model;  @{  string appTitle = "Asp.Net Core Demo App";  List<Person> people = new List<Person>()  {  new Person()  {  Name = "Durga", DOB = DateTime.Parse("1989-11-24"), Gender = Gender.male  },  new Person()  {  Name = "Reshma", DOB = DateTime.Parse("1985-10-05"), Gender = Gender.female  },  new Person()  {  Name = "Rudra", DOB = DateTime.Parse("2024-10-06"), Gender = Gender.female  }  };  }  <!DOCTYPE html>  <html>  <head>  <title>@appTitle</title>  <meta charset="utf-8"></meta>  </head>  <body>  @foreach(Person p in people)  {  <div>  @p.Name, @p.DOB.ToString(), @p.Gender  </div>  }  </body>  </html> |
| --- |

**Razor - for**

For loop is used when you want to display data only for some number of time like you have collection of 5 elements and you want to display only 2 elements then you can use for loop

**@for(initialization,condition,increament)**

**{**

**C#/HTML Code**

**}**

**Example**

| @for(int i = 0; i < 2; i++)  {  Person p = people[i];  <div>  @p.Name, @p.DOB.ToString(), @p.Gender  </div>  } |
| --- |

**Razor - Local Functions**

Local functions are callable methods that can accept some arguments and do some processing and return output. These local functions are accessible in the same view. For example when you have some repeated code in view then you can think of using them. You can use the local function inside the method, in the controller or in view as well. Scope of local function is limited to its containing method.

| @using ViewsExample.Model;  @{  string appTitle = "Asp.Net Core Demo App";  List<Person> people = new List<Person>()  {  new Person()  {  Name = "Durga", DOB = DateTime.Parse("1989-11-24"), Gender = Gender.male  },  new Person()  {  Name = "Reshma", DOB = DateTime.Parse("1985-10-05"), Gender = Gender.female  },  new Person()  {  Name = "Rudra", DOB = DateTime.Parse("2024-10-06"), Gender = Gender.female  }  };    }  @functions{  //local funciton  double? GetAge(DateTime? dateOfBirth)  {  if (dateOfBirth is not null)  {  return Math.Round((DateTime.Now - dateOfBirth.Value).TotalDays / 365);  }  else  {  return null;  }  }  }  <!DOCTYPE html>  <html>  <head>  <title>@appTitle</title>  <meta charset="utf-8"></meta>  </head>  <body>  @for(int i = 0; i < 2; i++)  {  Person p = people[i];  <div>  @p.Name,  @if(p.DOB != null)  {  <span>@p.DOB.Value.ToString("MM/dd/yyyy")</span>  <span>@GetAge(p.DOB)</span>  },  @p.Gender  </div>  }  </body>  </html> |
| --- |

**ViewData**

View Data is a dictionary object which is used to send data from controller to view. View data object automatically created upon receiving data from controller and deleted before sending response to client.

| **Controller**  using Microsoft.AspNetCore.Mvc;  namespace ViewsExample.Controllers  {  public class HomeController : Controller  {  [Route("/")]  public IActionResult Index()  {  ViewData["AppTitle"] = "Asp.Net Core Demo App";  return View();  }  }  }  **View**  <title>@ViewData["AppTitle"]</title> |
| --- |

**View With Collection Data**

| **Controller**  using Microsoft.AspNetCore.Mvc;  using ViewsExample.Model;  namespace ViewsExample.Controllers  {  public class HomeController : Controller  {  [Route("/")]  public IActionResult Index()  {  ViewData["AppTitle"] = "Asp.Net Core Demo App";  List<Person> people = new List<Person>()  {  new Person()  {  Name = "Durga", DOB = DateTime.Parse("1989-11-24"), PersonGender = Gender.male  },  new Person()  {  Name = "Reshma", DOB = DateTime.Parse("1985-10-05"), PersonGender = Gender.female  },  new Person()  {  Name = "Rudra", DOB = DateTime.Parse("2024-10-06"), PersonGender = Gender.female  }    };  ViewData["people"] = people;  return View();  }  }  }  **View**  @using ViewsExample.Model;  <!DOCTYPE html>  <html>  <head>  <title>@ViewData["AppTitle"]</title>  <meta charset="utf-8"></meta>  </head>  <body>  <h1>Welcome</h1>  @{  List<Person>? people = (List<Person>?)ViewData["people"];  }    @foreach (Person person in people!)  {  <p>  <span>@person.Name</span>  <span>@person.DOB.Value.ToString("MM/dd/yyyy")</span>  <span>@person.PersonGender</span>  </p>  }  </body>  </html> |
| --- |

**ViewBag**

View bag is a property of controller and view that is used to access the view data easily (without type casting). It is not separate from viewdata but it internally uses viewdata to retrieve data. Also it is best practice to use type of object rather than using var keyword in for each loop as type of variable is decided at runtime so while writing code it does not show any intelices for properties of some class for example as well as we need to handle null exception while using view bag.

| **View**  @using ViewsExample.Model;  <!DOCTYPE html>  <html>  <head>  <title>@ViewData["AppTitle"]</title>  <meta charset="utf-8"></meta>  <link href="~/StyleSheet.css" rel="stylesheet"/>  </head>  <body>  <div class="page-content">  <h1 class="header">Welcome</h1>  @{  //List<Person>? people = (List<Person>?)ViewData["people"];  }  @foreach (Person person in ViewBag.people)  {    <div class="box">  <span>@person.Name</span>  <table class="table w-100">  <tbody>  <tr>  <td>Date Of Birth</td>  <td><span>@person.DOB.Value.ToString("MM/dd/yyyy")</span></td>  </tr>  <tr>  <td>Gender</td>  <td><span>@person.PersonGender</span></td>  </tr>  </tbody>  </table>  </div>  }  </div>  </body>  </html> |
| --- |

**Strongly Typed View**

Strongly typed view is a view that is bound to a specific model class. It is mainly used to access the model/object collection easily in the view. To specify the model you need to write

@model ObjectName like that you need to mention. While passing data from controller to view you need to pass objects in view while returning.

Ex return View(objectName);

| **Controller**  public IActionResult Index()  {  ViewData["AppTitle"] = "Asp.Net Core Demo App";  List<Person> people = new List<Person>()  {  new Person()  {  Name = "Durga", DOB = DateTime.Parse("1989-11-24"), PersonGender = Gender.male  },  new Person()  {  Name = "Reshma", DOB = DateTime.Parse("1985-10-05"), PersonGender = Gender.female  },  new Person()  {  Name = "Rudra", DOB = DateTime.Parse("2024-10-06"), PersonGender = Gender.female  }  };  return View(people);  } |
| --- |
| **View**  @using ViewsExample.Model;  @model IEnumerable<Person>;  <!DOCTYPE html>  <html>  <head>  <title>@ViewData["AppTitle"]</title>  <meta charset="utf-8"></meta>  <link href="~/StyleSheet.css" rel="stylesheet"/>  </head>  <body>  <div class="page-content">  <h1 class="header">Welcome</h1>  @{  //List<Person>? people = (List<Person>?)ViewData["people"];  }  @foreach (Person person in Model)  {  <div class="box">  <span>@person.Name</span>  <table class="table w-100">  <tbody>  <tr>  <td>Date Of Birth</td>  <td><span>@person.DOB!.Value.ToString("MM/dd/yyyy")</span></td>  </tr>  <tr>  <td>Gender</td>  <td><span>@person.PersonGender</span></td>  </tr>  </tbody>  </table>  </div>  }  </div>  </body>  </html> |

**Benefits of Strongly Typed View**

* You will get intellisense while accessing model class properties
* You will have only one model per view
* Property name checked at compile time
* Easy to identify which model is being accessed in the view

**ViewMode**

View model is also called a wrapper model that is used to combine multiple entity classes and display results into strongly typed views. Strongly typed view is assigned with one model class only to bind multiple model classes to a strongly typed view. We can use ViewModel. It is nothing but creating one class which is considered as view model class add other class as property into it and bind it with a view that's it.

**\_ViewImports.cshtml**

\_ViewImports.cshtml is a special file in the Views folder or its subfolder which executes automatically before execution of a view. It is mainly used to import common namespaces that are imported in a view.

**SharedViews**

In a real world scenario we come across a situation where we want to have the same UI components across multiple controllers and view that part we can keep it in the shared view folder for example menu, footer etc will come under SharedView.

**Layout View**

Layout view is a web page that is responsible for containing presentation logic templates that are common like header, sidebar, footer etc. To create layout we usually have a shared folder in views then we can add a new item as Razor Layout then give a specific name which usually starts with underscore. @RenderBody() method is responsible for rendering content from other views. When we want to use layout view in other view at start we need to import it

@{

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

}

Both View and Layout View share the same view data object so transferring data is also possible here. And If we apply css/js files to Layout view then it is automatically applicable to other views.

**\_ViewStart.cs**

\_ViewStart.cs is a file where you can import your layout view then if you place it in the views folder it will be automatically applicable to all Views. Even you can specifically place it in a particular view folder like about, contact etc but then it is applicable to the view contained into that particular folder. This file provides a centralised approach for importing layout views.

**Dynamic Layout View**

Dynamic Layout view is used to load different layout views as per view. We can load Layout view in other views dynamically

| @{  ViewData["Title"] = "Search";  if(ViewBag.ProductID != null)  {  Layout = "~/Views/Shared/\_ProductsLayout.cshtml";  }  }  <h1>Search</h1>  <div>  <p>Search product logic comes here</p>  </div> |
| --- |

**Layout View Section**

When you want to display some particular section of the layout view based on normal views you can write that code inside of the section. While rendering a section in LayoutView we need to make sure that name should be matching with the section name of source view.

| **View**  @{  ViewData["Title"] = "Contact";  }  <h1>Contact Us</h1>  <div>  <table>  <tr>  <td>Contact No</td>  <td>xxx-xxx-xxx</td>  </tr>  <tr>  <td>Email</td>  <td>support@test.in</td>  </tr>  <tr>  <td>Address</td>  <td>Lorem Ipsum , Lorem Ipsum , Lorem 000977</td>  </tr>  </table>  @section secfooter  {  <p>Office time 9 AM to 6 AM Mon - Fri</p>  <p>Sat Sun Holiday</p>  }  </div>  **LayoutView**  <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>@ViewData["Title"]</title>  <link href="~/StyleSheet.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <div class="navbar">  <div class="navbar-brand">Layout Demo App</div>  <ul>  <li><a href="/">Home</a></li>  <li><a href="/about-company">About</a></li>  <li><a href="/contact">Contant</a></li>  <li><a href="/products">Products</a></li>  <li><a href="/search-product">Search Products</a></li>  <li><a href="/order-product">Order</a></li>  </ul>  </div>  <div class="page-content">  @RenderBody()  </div>  </div>  <div class="footer\_content">  @RenderSection("secfooter", required: false)  </div>  </body>  </html> |
| --- |

**Partial Views**

Partial View is a razor markup file that can’t be invoked individually from the controller but can be invoked from any view within the same application.