Unit 4
ASP.NET Core MVC Application

MVC Pattern

- In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
- It is a description or template for how to solve a problem that can be used in many different situations
- It does not belong to specific programming language or framework, but it is a concept that you can use in creating any kind of application or software in any programming language
- Model–view–controller (usually known as MVC) is a one of the software design pattern commonly used for developing applications that divides the related program logic into three interconnected components i.e. model, view and controller, as illustrated below.

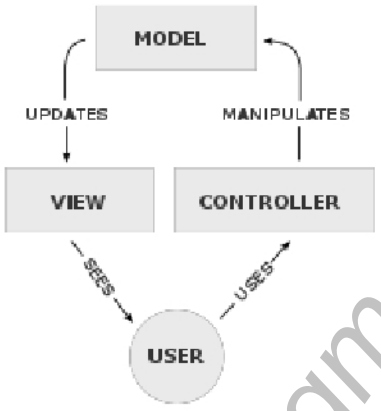


Fig: Interaction in MVC Pattern

1. Model and its Responsibility
   - Also called as application logic layer or domain layer.
   - The model is responsible for managing the data of the application.
   - It responds to the request from the view and it also responds to instructions from the controller to update itself.
   - Model are the classes that represent the data of the application and that use validation logic to enforce business rules for that data.
   - It is the part of the application that handles the logic for the application data.
   - Model works directly with the database. It does not have to deal with user interface or data processing. In real world scenario, we simply use model to fetch, insert, update and delete data from the database.
2. View and its responsibility
   - It means presentation of data like database record, in a particular format, triggered by a controller's decision to present the data.
   - Views renders the model into a form which is suitable for interaction, typically a user interface element like HTML page. So, simply view is the User Interface on which user can perform some actions. It contains HTML, CSS, JS, XML or any other markup language that we can use to create a beautiful user interface.
   - Multiple views of the same data are possible.
   - It is the parts of the application that is responsible for presenting/displaying the content/data through the user interface.
   - ASP.NET MVC Core uses Razor View Engine to embed .NET code in HTML markup.
3. Controller and its responsibility
   - The controllers are the components are responsible for user interaction, work with the data model, and ultimately select a view to render
   - The controller receives the input, it validates the input and then performs the business operation that modifies the state of the data model. So, controller is the part in which we process the data after we get a request from View and before updating anything in our database with our Model.
   - In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).
   - Example: Controller are the classes that handle incoming browser requests (user interaction), retrieve model data, and then specify view templates that return a response to the browser

- MVC is popular as it isolates the application logic from the user interface layer and supports **separation of concerns**.
- The ASP.NET Core MVC is a web application framework developed by Microsoft, which implements the model–view–controller pattern. Similarly, Laravel in Php, Spring in java etc.

Advantages of MVC architecture:

1) Separation on Concern.
2) Development of the application becomes fast.
3) Easy for multiple developers to collaborate and work together.
4) Easier to Update the application.
5) Easier to Debug as we have multiple levels properly written in the application.

Disadvantages of MVC architecture:

1) Understanding flow of application is very hard one. It is little bit of complex to implement and not suitable for small level applications.
2) Must have strict rules on methods.

ASP.NET Core MVC

- The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.
- ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns using MVC design pattern.
- **Major Features of ASP.NET Core MVC includes**
  1. Routing
  2. Model Binding
  3. Model Validation
  4. Strongly Typed Views
  5. Tag Helpers
  6. Razor View Engine
  7. IoC Container and Dependency Injection
  8. Web APIs
  9. Area //Will Be discussed in coming chapter
  10. Filter ////Will Be discussed in coming chapter

Basic Terminologies in ASP.NET Core Development Enviroment

1. **Middleware**
   - Middleware is a piece of software that handles HTTP requests and responses in the middle of application pipeline
   - ASP.NET Core implements MVC using a single piece of middleware, which is normally placed at the end of the middleware pipeline, as shown in figure below.

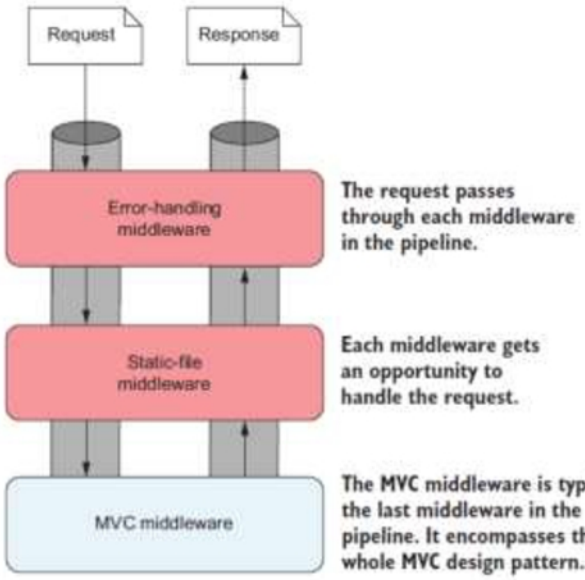


Fig: Middleware Pipeline

- In the above middleware pipeline, the request passes through the error handler middleware and into the static file middleware and finally to MVC middleware.
- Exception Handler Middleware is used to display a friendly customized error page to the end users. Also, Developer Exception page middleware can be added instead of this to t to throw exceptions and details of current requests to assist developers in better error diagnosis.
- The static file middleware returns the requested CSS file, image and other resources as the response, which passes back to the web server.
- A middleware component may handle the request and decide not to call the next middleware in the pipeline. This is called **short-circuiting** the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, if the request is for a static file like an image or css file, the StaticFiles middleware can handle and serve that request and short-circuit the rest of the pipeline. This means in our case, the StaticFiles middleware will not call the MVC middleware if the request is for a static file.

2. Startup Class

- ASP.NET Core apps use a Startup class, which is named Startup by convention.
- The startup class consists of two important methods. These methods are called by the ASP.NET Core runtime when the app starts:
  1. ConfigurationServices
     - The Startup class: Optionally includes a ConfigureServices method to configure the app's services.
     - A service is a reusable component that provides app functionality. Services are registered in ConfigureServices and consumed across the app via dependency injection (DI).
     - For features that require substantial setup, there are Add{Service} extension methods on IServiceCollection. For example, **Add**Authorization, **Add**Authentication, **Add**Logging and **Add**Mvc etc.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddAuthorization();
        services.AddAuthentication();
        services.AddLogging();
        services.AddMvc();
}
```

  2. Configure
     - In ASP.NET Core, Middleware is a piece of software that can handle an HTTP request or response.
     - A given middleware component in ASP.NET Core has a very specific purpose. For example we may have a middleware component that authenticates a user, another piece of middleware to handle errors, yet another middleware to serve static files such as JavaScript files, CSS files, Images etc.

- The request pipeline is configured as part of the application startup by the Configure() method in Startup.cs file.
- The following is the code in Configure() method.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
        if (env.IsDevelopment())
        {
                app.UseDeveloperExceptionPage();
        }
        else
        {
                app.UseExceptionHandler("/Home/Error");
        }
        app.UseStaticFiles();
        app.UseMvc(routes =>
        {
                routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}");
        });  //Default Template Routing Middleware
}
```

- The above Configure () method sets up a very simple request processing pipeline with just three pieces of middleware

Controller, Action and Routing

- Controllers are essentially the central unit of your ASP.NET MVC application.
- It is the first recipient, which interacts with incoming HTTP Request. So, the controller decides which model will be selected, and then it takes the data from the model and passes the same to the respective view, after that view is rendered.
- Actually, controllers are controlling the overall flow of the application taking the input and rendering the proper output.
- Controllers are simple C# classes inheriting from System.Web. Mvc.Controller, which is the built-in controller base class. So, a controller can have fields, properties and methods.
- Each public method in a controller is known as an **action method,** meaning you can invoke it from the Web via some URL to perform an action. For this, we use concept called **Routing.**
- Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints i.e. action methods on our controller. Endpoints are the app's units of executable request-handling code.
- We can change the routing at the application level, controller level, and action level.
- The MVC convention is to put controllers in the Controllers folder that Visual Studio created when the project was set up.

Example:

```
using Microsoft.AspNetCore.Mvc;
namespace MyApplication.Controllers
{
        public class HomeController : Controller // Controller Home
        {
                public IActionResult Index()//Action Method Index
                {
                        return View();
                }
        }
}
```

A URL http://localhost/Home/Index will invoke Index Action method of Home Controller

Action Verb

- The ActionVerbs selector is to handle different type of Http requests.
- Some of the action verbs included by the MVC framework includes HttpGet (To retrieve the information from the server. Parameters will be appended in the query string.), HttpPost (To create a new resource.), HttpPut (To update an existing resource.), HttpDelete(To delete an existing resource.), HttpDelete(To delete an existing resource).
- To gain more control of how our actions are called, we can decorate action methods with Action verb.
- We can apply one or more action verbs to an action method to handle different HTTP requests.
- If you don't apply any action verbs to an action method, then it will handle HttpGet request by default.

Example:

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



        [HttpGet]
        public IActionResult ShowAllRecords()
        {
            return View();
        }


        [HttpPost]
        public IActionResult SaveRecords()
        {
            return View();
        }

        [HttpGet]
        [HttpPost]
        public IActionResult MyAction()
        {
            return View();
        }
    }
}
```

In the above, the Index() action can be accessed by Get() request by default, ShowAllRecords() via Get Request SaveRecords() via Post Request and MyAction via both Get and Post request. Also we can define two action methods within same controller by decorating them with different action verb as below

```
        [HttpGet]
        public IActionResult Edit() // display Form for editing some item
        {
            return View();
        }
        [HttpPost]
        public IActionResult Edit()// update the item detail and save
        {
            return View();
        }
```

Action Result Type

- Action Result is actually a data type.
- When it is used with action method, it is called return type.
- As we know, an action is referred to as a method of the controller, the Action Result is the result of action when it executes.

- When the Action finishes its work, it will usually return something to the client and that something will be usually implemented by IActionResult Interface.
- ActionResult is the default implementation of IActionResult Interface.
- Action Result is a base data type and some of its derived types that can be used for generating action results includes

1. ContentResult: returns the specified string as plain text to the client

```
public ContentResult Index()
{
        return Content("Hello World");
}
```

2. JsonResult

```
        public JsonResult Index()
        {
              return Json(new { id=1, name="ram"}
        }
```

3. ViewResult
   ViewResult is a datatype which is responsible for returning the regular View to the client.

```
        public ViewResult Index()
        {
               return View();
        }
```

4. PartialViewResult
   It is the type which is used to return the partial view page rather than returning the regular view page. It used to return the portion of web page instead of whole page.

```
public PartialViewResult Index()
{
        return PartialView();
}
```

5. RedirectResult
   If we use this type, then it'll redirect to the URL specified by us.

```
        public RedirectResult Index()
        {
               return Redirect("https://www.facebook.com");
        }
```

6. RedirectToRouteResult
   It is responsible for the redirection to the actions within the application We use RedirectToRoute which redirect us to the action within the specified controller.

```
        public RedirectToRouteResult Index()
        {
              return RedirectToRoute(new { controller = "Student", action = "Name" });
        }
```

   The above action method upon execution will redirect to Name action method within the Student controller. i.e. /Student/Name.

7. RedirectToActionResult
   If we are in the same controller and don't want to give the name of controller then the better option of is, RedirectToAction helper method.

```
        public RedirectToActionResult Index()
        {
               return RedirectToAction("About");
        }
```

Here, the action method About is in the same controller as that of Index action method.

8. StatusCodeResult
   This type is used to give HTTP status code like unauthorized access (401), NotFound (404) etc. to the client.

```
public StatusCodeResult Index()
{
     return StatusCode(401);
}

public StatusCodeResult Index()
{
       return  NotFound();
}

public StatusCodeResult Index()
{
       return Unauthorized();
}
```

Note:
As ActionResult is the base class for all response type, so we can use this type to return any response type.

```
public ActionResult Index()
{
       int a = 5;
       if (a % 2 == 0)
       {
              return Json("Number is Even");
       }
       else
       {
              return Content("<h1>Number is Odd<h1>","text/html");
       }
}
```

In the above code, if the number is even it will return a string message in json format but if the number is odd it will return a string message formatted as html as indicated by MIME (Multipurpose Internet Mail Extensions) type i.e. "text/html' in Content helper method to tell the MVC to take appropriate action after recognizing the content.

URL Routing

- The Routing in ASP.NET Core MVC application is a mechanism in which it will inspect the incoming Requests (i.e. URLs) and then mapped that request to the controllers and their action methods.
- This mapping is done by the routing rules which are defined for the application.
- We can do this by adding the Routing **Middleware** to the request processing pipeline.
- So, the ASP.NET Core Framework maps the incoming Requests i.e. URLs to the Controllers action methods based on the routes configured in your application.

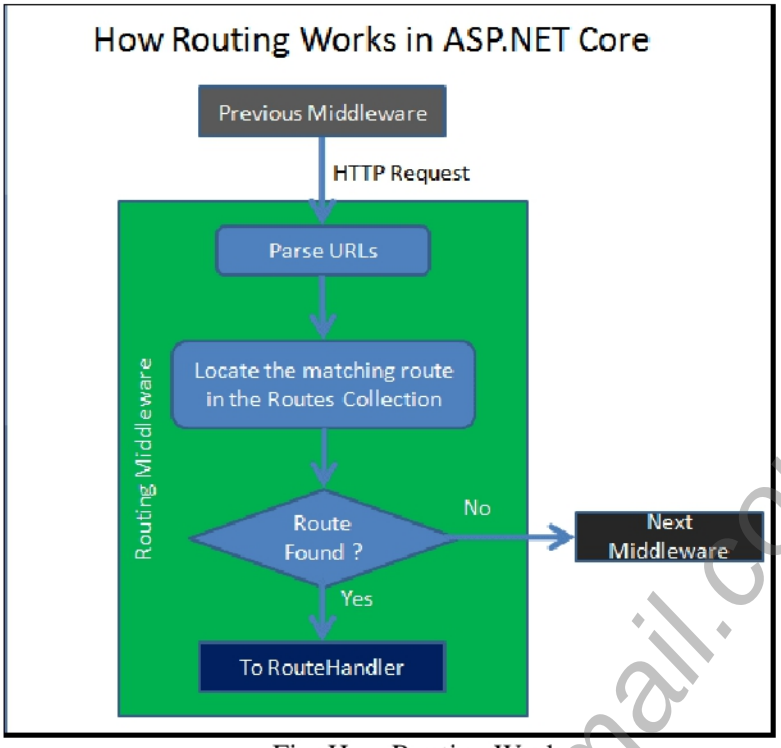


Fig: How Routing Works

- When the Request arrives at the Routing Middleware it does the following.
  1. It Parses the URL.
  2. Searches for the Matching Route in the RouteCollection.
  3. If the Route found, then it passes the control to RouteHandler.
  4. If Route not found, it gives up and invokes the next Middleware.
- Route:

  Each Route contains a Name, URL Pattern (Template), Defaults and Constraints. The URL Pattern is compared to the incoming URLs for a match. An example of URL Pattern is {controller=Home}/{action=Index}/{id?}
- Route Collection

  The Route Collection is the collection of all the Routes in the Application. The Routing Module looks for a Route that matches the incoming request URL on each available Route in the Route collection.

- Route Handler

  The Route Handler is the Component that decides what to do with the route.
- We can configure **multiple routes** for our application and for each route you can also set some specific configurations such as default values, constraints etc.

- ASP.NET Core MVC application, you can define routes in two ways. They are as follows:
  1. **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 Configure method of the Startup.cs class file.

  Example:
     - In ASP.NET Core MVC application, it is the controller action method that is going to handle the incoming Requests i.e. URLs. If we issue a request to the "/**Home/Index/2**" URL, then it is the Index(int id) action method of Home Controller class which is going to handle the request as shown in the below image.

```
http://localhost:52190/Home/Index/2

public class HomeController : Controller
    {
     public IActionResult Index(int id)
    {
                    return View();
```

```
        }
    }
```

- As we have not explicitly defined any routing rules for the application, then how does this mapping is done i.e. how the "**/Home/Index/2**" URL is mapped to the Index action method of the Home Controller class. Here the parameter value 2 is automatically mapped to the id parameter of the Details action method.
- This is actually done by the MVC Middleware which we registered in the application's request processing pipeline.
- We can add the required MVC middleware into the request processing pipeline either by calling the **UseMvcWithDefaultRoute()** method or by calling the **UseMvc()** method within in the Configure() method of the **Startup.cs** class file as shown in the below image.
  - UseMvcWithDefaultRoute() extension method adds MVC with the following default route to our application's request processing pipeline.

    {controller=Home}/{action=Index}/{id?}
  - We can define the equivalent routing information in more flexible way using UseMvc() middleware as defined as

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

  - For a request with URL http://localhost:52190/Home/Index/2, the first URL path segment "/Home" is mapped to the "HomeController".
  - The second URL path segment "/Index" is mapped to the Index(int id) action method in the HomeController.
  - The third path segment "1" is mapped to the id parameter of the Index(int id) action method. This is done by a process called **model binding**.
  - Notice in the following default route template, we have a question mark after the id parameter.

    The default route template "{controller=Home}/{action=Index}/{id?}"
  - The question mark, makes the id parameter in the URL optional. This means both the following URLs will be mapped to the Index() action method of the HomeController class.
  - Note, all the following URL are mapped to same action method Index(int id) of Home Controller.
    a. http://localhost:52190/Home/Index/3
    b. http://localhost:52190/Home/Index
    c. http://localhost:52190 (Default controller is Home and Default action is index)
- If you want to define your own route templates and want to have more control over the routes, use UseMvc() method, instead of UseMvcWithDefaultRoute() method.

Note:

- We can register multiple custom routes with different names. Consider the following example where we register "Admin" route and "Deault" route

```
app.UseMvc(routes =>
            {
                //routes.MapRoute("admindef", "{controller}/{action}/{id?}",new { Controller =
"Admin", Action = "Index" });
                //or
                //routes.MapRoute(name:"admindef", template: "{controller}/{action}/{id?}",
defaults: new { Controller = "Admin", Action = "Index" });
                //or
              //  routes.MapRoute(name: "admindefault", template:
                                "{controller=Admin}/{action=Index}/{id?}");

                // lets try some different then above
```

```
routes.MapRoute(name:"admindef", template: "Admin/{id?}", defaults: new {
        Controller = "Admin", Action = "Index" });

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

- In the first routing configuration, we have setup default values for controller and action method such that the controller and action defaults to Index Method of Admin Controller.

URL	Match?	Processed By	Passed As
https://localhost:44362/	Yes	Route-2	Controller: Home, Action: Index
https://localhost:44362/Admin	Yes	Route-1	Controller: Admin, Action: Index
https://localhost:44362/Home	Yes	Route-2	Controller: Home, Action: Index

- Order in which routes are registered is very important. URL matching starts ate the top and enumerates the collection of Routes searching for a match. It stops when it finds the first match.

2. Attribute-Based Routing.

- In Attribute-Based Routing, the route is determined based on the attributes which are configured either at the controller level or at the action method level.
- We can use both Conventional Based Routing and Attribute-Based Routing in a single application.
- With the help of ASP.NET Core Attribute Routing, you can use the Route attribute to define routes for your application.
- When you apply the Route attribute at the Controller level, then it is applicable for all the action methods of that controller.

Example

- Consider we have not configured any routes in configure() method of startup.cs class i.e. UseMvcWithDefaultRoute() middleware not configured and similarly usemvc() extension method is configured but without default routing template as below.

```
app.UseMvc();
```

- Now when we navigate to any of the following URLs we see 404 errors.
  - http://localhost:52190/Home/Index/3
  - http://localhost:52190/Home/Index
  - http://localhost:52190
- With attribute routing, we use the Route attribute to define our routes. Consider the following code snipshot.

```
public class HomeController : Controller
{
        [Route("Home/Index/{id?}")]
        [Route("Home/Index")]
        [Route("Home")]
        [Route("")]
        public IActionResult Index(int id)
        {
            return View();
        }
}
```

- Now when we navigate to any of the following URLs we see it is mapped to Index(int id) action method of Home Controller
  - http://localhost:52190/Home/Index/3
  - http://localhost:52190/Home/Index
  - http://localhost:52190/Home
  - http://localhost:52190
- If we notice, we have repeated the word Home multiple times. In order to make these routes less repetitive, we need to apply the Route() attribute with the word Home at the HomeController class level as shown

below. So to make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual action methods.

```
[Route("Home")]
[Route("")]
public class HomeController : Controller
{
        [Route("Index/{id?}")]
        [Route("Index")]
        [Route("")]
        public IActionResult Index(int id)
        {
            return View();
        }
}
```

- The Route template applied on the controller level is prepended to the route template applied to the action method level.
- Now when you navigate to the following four URLs you will get the output as expected.

  - http://localhost:52190/Home/Index/3
  - http://localhost:52190/Home/Index
  - http://localhost:52190/Home
  - http://localhost:52190

Rendering HTML with Views

1. In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction.
2. A view is an HTML template with embedded Razor markup.
3. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.
4. In ASP.NET Core MVC, views are *.cshtml* files that use the C# programming language in Razor markup.
5. Usually, view files are grouped into folders named for each of the app's controllers.
6. The folders are stored in a *Views* folder at the root of the app
7. The below flow chart shows how a particular html view request is processed.

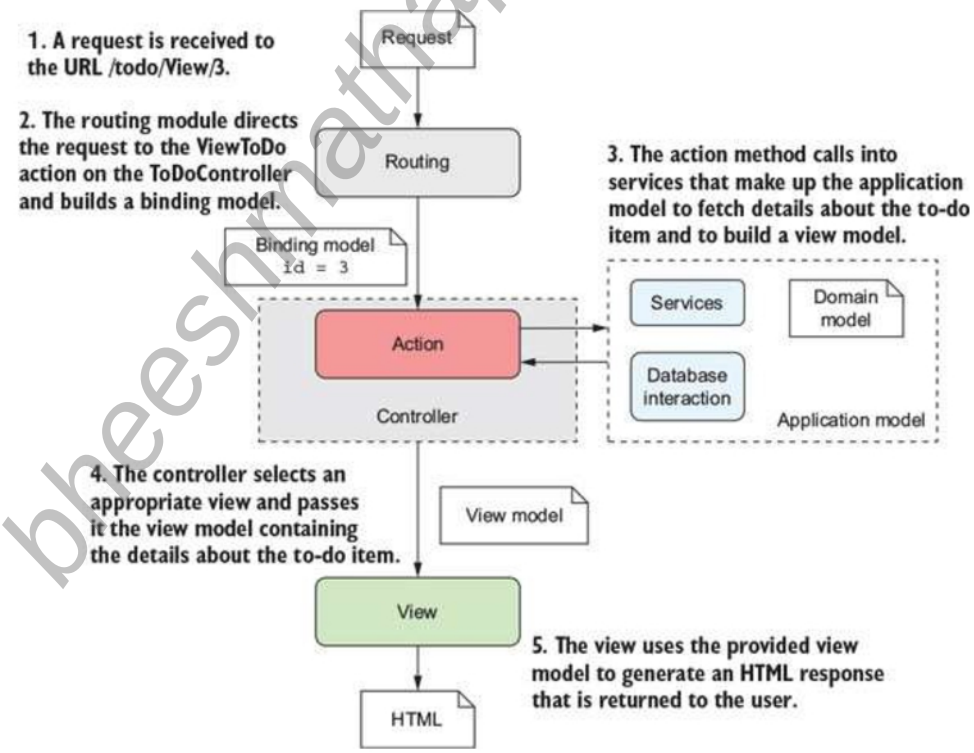


Fig: Steps to process HTML View Request

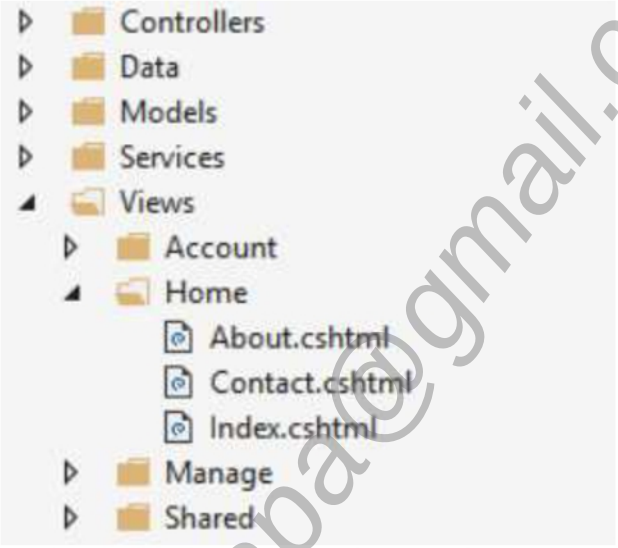
Benefits of using views

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

1. The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
2. The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
3. It's easier to test the user interface parts of the app because the views are separate units.
4. Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating View

- Views that are specific to a controller are created in the *Views/[ControllerName]* folder.
- Views such as Layout and Partial View that are shared among controllers are placed in the *Views/Shared* folder.
- To create a view, add a new file and give it the same name as its associated controller action with the *.cshtml* file extension.
- To create a view that corresponds with the *About* action in the *Home* controller, create an *About.cshtml* file in the *Views/Home* folder as shown in figure below.



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

Example: The following Program shows how a view is displayed

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

```
<h1> This is My First View </h1>
```

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

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

Output

- Views are typically returned from actions as a ViewResult, which is a type of ActionResult. Our action method can create and return a ViewResult directly, but that isn't commonly done. Since most controllers inherit from Controller, you simply use the View helper method to return the ViewResult:

Defining Views with Razor syntax

- Razor is a view engine for MVC framework.
- Razor is a markup syntax that lets you embed server-based code (Visual Basic and C#) into web pages.
- Server-based code can create dynamic web content on the fly, while a web page is written to the browser.
- When a web page is called, the server executes the server-based code inside the page before it returns the page to the browser. By running on the server, the code can perform complex tasks, like accessing databases.
- It has the power of traditional ASP.NET (.ASPX) markup, but it is easier to use, flexible and lightweight.
- Unlike traditional .ASPX view engine, the razor view engine offers one biggest advantage which is we can mix client side markup html and server side code c#, without having to jump in and out of the two syntax type.
- Just like in regular c# code, in Razor we can define variable, use decision making statements like if, switch, looping statements like for loop, for each loop, define functions and call it etc.
- Razor markup starts with the @ symbol within in html document.
- We can run set of C# statements by placing C# code within Razor code blocks set off by curly braces ({ ... }).
- Similarly, we can display values within HTML by simply referencing the value with the @ symbol.

Example:

WAP to demonstrate how server side code(C#) can be combined with html in razor syntax.

(Display your name 10 times)

```
<html>
<body>
<h1>Name</h1>
@for (int i = 0; i < 10; i++)
{
<h4>Hello World</h4>
}
<h1>Thank You.</h1>
</body>
</html>
```

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

```
<html>
<body>
@for (int i = 0; i < 100; i++)
{
    if (i % 2 == 0)
    {
```

```
            <h2 style="color:green">@i</h2>
        }
        else
        {
            <h2 style="color:red">@i</h2>
        }
    }
    </body>
    </html>
```

HTML Helper

- An HTML helper is a method that is used to render html content in a view. HTML helpers are implemented as extension methods.
- HTML helpers will greatly reduce the amount of HTML that we have to write in a view. Views should be as simple as possible. All the complicated logic to generate a control can be encapsulated into the helper, to keep views simple.
- For example,
  1. **To produce the HTML for a textbox with id="firstname" and name="firstname", we can type all the** html in the view as shown below

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

OR

We can use the "TextBox" html helper.

@Html.TextBox("firstname")

There are several overloaded versions. To set a value, along with the name, use the following overloaded version.

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

The above html helper, generates the following HTML

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

To set HTML attributes, use the following overloaded version. Notice that, we are passing HTML attributes (style & title) as an anonymous type.

@Html.TextBox("firstname", "John", new { style = "background-color:Red; color:White; font-weight:bold", title="Please enter your first name" })

Some of the html attributes, are reserved keywords. Examples include class, readonly etc. To use these attributes, use "@" symbol as shown below.

@Html.TextBox("firstname", "John", new { @class = "redtextbox", @readonly="true" })

Similarly,

@Html.ActionLink("Create New", "Create")

would generate anchor tag

Create New.

- There are many extension methods and Strongly Typed Method for HtmlHelper class, which can be used to creates different HTML controls as summarized below.

Extension Method	Strongly Typed Method	Html Control
@Html.BeginForm("Index","Home", FormMethod.Post){}	NA	<form method="post" action="/Home/Index"></form>
Html.ActionLink()	NA	<a>
Html.TextBox()	Html.TextBoxFor()	<input type="textbox">
Html.TextArea()	Html.TextAreaFor()	<input type="textarea">
Html.CheckBox()	Html.CheckBoxFor()	<input type="checkbox">
Html.RadioButton()	Html.RadioButtonFor()	<input type="radio">
Html.DropDownList()	Html.DropDownListFor()	<select> <option> </select>
Html.ListBox()	Html.ListBoxFor()	multi-select list box: <select>
Html.Hidden()	Html.HiddenFor()	<input type="hidden">

Extension Method	Strongly Typed Method	Html Control
Html.Password()	Html.PasswordFor()	<input type="password">
Html.Display()	Html.DisplayFor()	HTML text: ""
Html.Label()	Html.LabelFor()	<label>
Html.Editor()	Html.EditorFor()	Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type.

2. **Consider a Razor view with the following model:**

```
public class Movie
{
   public int ID { get; set; }
   public string Title { get; set; }
   public DateTime ReleaseDate { get; set; }
   public string Genre { get; set; }
   public decimal Price { get; set; }
}
```

Defining View for the above model with Razor Markup using html helper method

```
@model WebApplication60.Models.Movie
@Html.LabelFor(m => m.ID)
@Html.TextBoxFor(m=>m.ID)
```

The above razor markup will generate the following equivalent html

```
<label for="ID">ID</label>
<input type=”text” name=”ID”/>
```

Tag Helper

- Tag Helpers are server side components. They are processed on the server to create and render HTML elements in Razor files.
- Tag Helpers are similar to HTML helpers. There are many built-in Tag Helpers for common tasks such as generating links, creating forms, loading assets etc.
- HTML helpers were methods accessed through Razor expressions that begin with @Html, for example. @Html.TextBoxFor(m => m.Name) would create a textbox for the name property. The problem with HTML helper expressions is that they don’t fit with the structure of HTML elements, which leads to awkward expressions, especially when adding CSS, for example. @Html.TextBoxFor(m => m.Name, new { @class = “form-control” }). Attributes have to be expressed in a dynamic object and have to be prefixed with @ if they are reserved C# words, like class. As the HTML elements that are required become more complex, the HTML helper expression becomes more awkward. Tag helpers remove this awkwardness, for example, <input class=”form-control” asp-for=”Name” />.
- The result is a more natural fit with the nature of HTML and produces views that are easier to read and understand. MVC still supports HTML helpers, so you can use them for backward compatibility in views originally developed for MVC 5, but new views should take advantage of the more natural approach that tag helpers provide.

Example:

3. **Generating Links using Tag Helpers**

Let's say we want to view a specific employee details. So we want to generate the following hyperlink.

The number 5 is the ID of the employee whose details we want to view.

There are many possible we could do this in razor view

View [Normal HTML]

or

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

Or

<a asp-controller="home" asp-action="details" [Using Tag Helpers]
asp-route-id="5">View

The Anchor Tag Helper enhances the standard HTML anchor (<a ... >) tag by adding new attributes such as
1. asp-controller

2. asp-action
3. asp-route-{value}

As the names imply asp-controller specifies the controller name and asp-action specifies the action name to include in the generated href attribute value. asp-route-{value} attribute is used to include route data in the generated href attribute value. {value} can be replaced with the route parameters such as id for example.

Note: Using Class and Style Attribute

a. In Tag Helper

Example:

```
<input asp-for="stdid" class="form-control" placeholder="Enter-id"
                                    style="width:500px;height:25px;color:red" />
```

a. In HtmlHelper

Example:

```
@Html.TextBoxFor(model => model.stdid, new { @class="form-control", placeholder = "Enter id", style
= "width:500px;height:25px;color:red" })
```

4. **Consider a Razor view with the following model:**

```
public class Movie
{
   public int ID { get; set; }
   public string Title { get; set; }
   public DateTime ReleaseDate { get; set; }
   public string Genre { get; set; }
   public decimal Price { get; set; }
}
```

Defining View for the above model with Razor Markup using tag helpers

```
@model WebApplication60.Models.Movie
<label asp-for="ID"></label>
<input asp-for="ID"/>
```

The above razor markup will generate the following equivalent html

```
<label for="ID">ID</label>
<input type=”text” name=”ID”/>
```

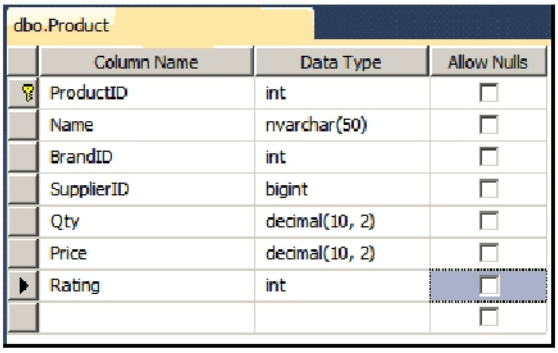
Advantage of Tag Helpers

- An HTML-friendly development experience for the most part, Razor markup using Tag Helpers looks like standard HTML.
- **A rich IntelliSense environment for creating HTML and Razor markup**

Model, Model Binding and Model Validation

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

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



dbo.Product

Column Name	Data Type	Allow Nulls
ProductID	int	☐
Name	nvarchar(50)	☐
BrandID	int	☐
SupplierID	bigint	☐
Qty	decimal(10, 2)	☐
Price	decimal(10, 2)	☐
Rating	int	☐
		☐

Fig: Product Table

```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }

    public Decimal Price { get; set; }
    public int Rating { get; set; }

    public Brand Brand { get; set; }
    public Supplier Supplier { get; set; }
}
```

Fig: Product Model

2. View Model:
    - The View Model refers to the objects which hold the data that needs to be shown to the user.
    - The View Model is related to the presentation layer of our application.
    - They are defined based on how the data is presented to the user rather than how they are stored.
    - The View Models can also have View related logic like displaying the Rating as "*" to the user as shown in figure below.
    - We should not put any other logic other than related to the display of the View in the ViewModel

```
public class ProductViewModel
{
    public int ProductId { get; set; }
    public string Name { get; set; }

    public Decimal Price { get; set; }
    public int Rating { get; set; }

    public string BrandName { get; set; }
    public string SupplierName { get; set; }

    public string getRating()
    {
        if (Rating == 10)
        {
            return "*****";
        }
        else if (Rating >=8 )
        {
            return "****";
        }
        else if (Rating >= 6)
        {
            return "***";
        }
        else if (Rating >= 4)
        {
            return "**";
        }
        else
        {
           return "*";
        }
    }
}
```

Fig: View Model

Benefits of View Model

i. View Model can be used to define a view that is not already contained by an existing model.
ii. View Model can be used to define a view by combining multiple models.
iii. View model can be used to define a simple view by defining it as a sub model of existing complex model.
iv. View model can be used to define a new view in flexible manner as we can easily extend existing model.

3. Edit Model:
    - The Edit Model or Input Model represents the data that needs to be presented to the user for modification/inserting.
    - The UI Requirement of Product for Editing can be different from the model required for Viewing.
    - For Example, In the Product Model list above, the user needs to be shown the list of Brands & Supplier, while they add/edit the Product. Hence our model becomes

```
public class ProductEditModel
{
    public int ProductId { get; set; }

    [Required(ErrorMessage = "Product Name is Required")]
    [Display(Name = "Product Name")]
    public string Name { get; set; }

    public Decimal Price { get; set; }
    public int Rating { get; set; }

    public List<Brand> Brands { get; set; }
    public List<Supplier> Suppliers { get; set; }

    public int BrandID { get; set; }
    public int SupplierID { get; set; }
}
```

Fig: Edit Model

- The 2 most important topics about Models are:
  1. Model Binding – a process of extracting data from HTTP request and providing them to the action method's arguments/parameter.
  2. Model Validation – a process to validate the Model properties so that invalid entries are not entered in the - database.

Model Binding

- Model binding in MVC maps HTTP request data to the parameters of the controller's action method.
- The parameter may either be of simple type like integers, strings, double etc. or complex types.
- When we look into the request life cycle of MVC, the request is first received by the routing and from the URL, MVC identifies the Controller and action method, using routing.
- Routing binds the values from HTTP request to action's parameter, using model binder.
- MVC binds the request data to the action parameter by parameter name.
- MVC will bind the action parameter from the various parts of the request.
- The model binding system:
  1. Retrieves data from various sources such as route data, form fields, and query strings.
  2. Provides the data to controllers and Razor pages in method parameters and public properties.
  - All the values from Form values, route data, and query strings are stored as the **name-value** pairs.
- Model binding is a very important feature in MVC. It helps us to bind the client data to method parameter.
- Following are the data sources in the order, which model binding looks through.
  1. Form values
     - These are the form data such as value from textbox, radiobutton, checkbox etc. which go with HTTP request, using POST method.
     - For example:
     - Using default routing conventions, we could define controller like this

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

```
        [HttpPost]
        public IActionResult Index(string fullname, string roll, string email)
        {
            return Content("Full Name =" + fullname + " Roll=" + roll + "
            Email Address=" + email);
        }
    }
}
```

- Let the view for Index Action Method be defined in Home Controller as

```
<form method="post" action="/Home/Index">
    Enter Your Name:
    <input type="text" name="fullname" />
    <br />
    Enter Your Roll:
    <input type="text" name="roll" />
    <br />
    Enter Your Email Address:
    <input type="text" name="email" />
    <br />
    <input type="submit" />
</form>
```

- So when the user presses submit button, the textbox "fullname" value in the form is mapped to the parameter fullname of the Index(string sname, string roll, string email) action method of Home Controller. Similarly roll and email are also mapped. The MVC framework automatic binding does this mapping. Be careful the name fullname, roll and email in the form field and the action method should be same for this automatic mapping.

2. Route values
   - The set of route values provided by MVC routing.
   - For example, Consider the action method is defined in Home Controller as below

```
using Microsoft.AspNetCore.Mvc;
namespace MyMvcApplication.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index(int id)
        {
            //Work with Id here passed as route value
            return View();
        }
    }
}
```

   - Again, consider the get request with URL, given below-
     https://localhost:44392/Home/Index/1
     in the above URL 1 is the route value.
   - Our application default route template as given below routes the above get request to the Index(int id) method of Home Controller.

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

   - So the value 1 in the request url is mapped to the id parameter of the Index(int id) action method of Home Controller. The MVC framework automatic binding does this mapping. Be careful the parameter id in the route template and the action method should be same for this automatic mapping.

Note:
If there are multiple parameter in the action methods

```
using Microsoft.AspNetCore.Mvc;
namespace MyMvcApplication.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index(int id, string name)
        {
            //Work with id  and name parameter here passed as route value
              return View();
        }
    }
}
```

Then, the route template is defined as

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

3. Query strings
   - The query string data, part of the URL.
   - When you see a URL, such as "http://www.example.com:88/home?item=book", it is comprised of several components. We can break this URL into 5 parts:
     - http: It tells the web client to use the Hypertext Transfer Protocol or HTTP to make a request. www.example.com: The **host**. It tells the client where the resource is hosted or located.
     - :88 : The **port** or port number. It is only required if you want to use a port other than the default.
     - /home/: The **path**. It shows what local resource is being requested.
     - ?item=book : The **query string**, which is made up of **query parameters**. It is used to send data to the server. This part of the URL is also optional.
   - For example, Consider the action method is defined in Home Controller as below

```
using Microsoft.AspNetCore.Mvc;
namespace MyMvcApplication.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index(int id, string name)
        {
            //Work with id  and name parameter here passed as route value
              return View();
        }
    }
}
```

   - Again, consider the get request with URL, given below-
     https://localhost:44392/Home/Index?id=2&name=siva
     in the above URL id and name are the query string.
   - Our application default route template as given below routes the above get request to the Index(int id) method of Home Controller.

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

- So the value 1 in the request url is mapped to the id parameter of the Index(int id, string name) action method of Home Controller. The MVC framework automatic binding does this mapping. Be careful the parameter id and name in the route template and the action method should be same for this automatic mapping.

Model Binding with Complex/User Defined Type

- In addition to simple binding discussed earlier, model Binding with Asp.Net Core also works with Complex type or user defined type like object.
- If the request is complex, i.e. we pass data in request body as an entity with the desired content-type, then such kind of request is mapped by Complex model binder.
- The following Example demonstrates the concept.

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

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

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

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

The @model directive lets the view know about which model it expects. Tag helper are the server side components that are processed on the server to create and render HTML elements. Examples of common built-in Tag Helper are Anchor tag, label tag, input tag etc.[*will be discussed in detail]*

Note: The above view created using tag helper will be converted to equivalent html as below

```
<form method="post" action="/Home/submit">
  <div>
    <label for="fullname">fullname</label>
    <input type="text" id="fullname" name="fullname" value="">
  </div>
  <div>
    <label for="roll">roll</label>
    <input type="text" id="roll" name="roll" value="">
  </div>
  <div>
```

```
    <label for="email">email</label>
    <input type="text" id="email" name="email" value="">
  </div>
  <div>
    <input type="submit" value="Submit">
  </div>
</form>
```

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

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

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

Model Validation using Data annotation attribute

- Model Validation is a process to ensure the data received from the View is appropriate to bind the Model. If it is not, then appropriate error messages are displayed on the View, and that will help user to rectify the problem.
- ASP.NET MVC include several built-in attribute classes in the System.ComponentModel.DataAnnotations for this purpose. Data annotations are used to add metadata for ASP.NET MVC and ASP.NET data controls.
- We can apply these attributes to the properties of the model class to validate and display appropriate validation messages to the users.
- Some of these attributes includes

1. [Required]: Specifies that a property value is required. This validates that the field is not null
   Example:
   ```
   [Required(ErrorMessage ="Full Name Must Be Provided")]
   ```

2. [StringLength]: Specifies the minimum and maximum length of characters that are allowed in a string type property. This validates that a string property value doesn't exceed a specified length limit.
   Example:
   ```
   [StringLength(50,MinimumLength =5,ErrorMessage ="Full Name must be within 5 to 50 characters")]
   ```

3. [Range]: Specifies the numeric range constraints for the value of a property. This validates that the property value within a specified range.
   Example:
   ```
   [Range(1,100, ErrorMessage ="Roll number must be in the range 1 to 100")]
   ```

4. [Compare]: This attribute validates that two properties in model class match like password and compare password.

Example:

```
[Compare("Password", ErrorMessage = "The password and confirm password do not
                                    match.")]
```

Where “Password” is another property to be compared with.

5. [RegularExpression]: Specifies that a property value must match the specified regular expression. This validates that the property value matches a specified regular expression.
   Example:

```
[RegularExpression("[a-z]{2}-[0-9]{4}-[0-9]{4}", ErrorMessage = "Data Must Be in the format xx-0000-
                                                0000")]
```

6. [EmailAddress]: This validates the property has email address format.
   Example:

```
[EmailAddress(ErrorMessage = "Invalid Email Address")]
```

The following Program Demonstrates the concept

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

```
using System.ComponentModel.DataAnnotations;
namespace WebApplication59.Models
{
    public class StudentViewModel
    {
        [Display(Name ="Full Name")]
        [Required(ErrorMessage ="Full Name Must Be Provided")]
        [StringLength(50,MinimumLength =5,ErrorMessage ="Full Name must be within 5 to
50 characters")]
        public string fullname { get; set; }
        [Range(1,100, ErrorMessage ="Roll number must be in the range 1 to 100")]
        [Required(ErrorMessage = "Roll Must Be Provided")]
        [Display(Name = "Roll Number")]
        public int roll { get; set; }
        [Required(ErrorMessage = "Email Address Must Be Provided")]
        [Display(Name ="Email Address")]
        [EmailAddress(ErrorMessage ="Please Provide Valid Email Address")]
        public string email { get; set; }
    }
}
```

2. **Now write a strongly typed html view that make use of the above model. Also we need to add three javascript libraries for client side validations. The validation-summary=”All” shows the summarized validation messages at the top.**

```
@model WebApplication59.Models.StudentViewModel
<html>
<head>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
</head>
<body>
    <form asp-action="/Submit" method="post">
        <div asp-validation-summary="All" style="color:red"></div>
        <div>
            <label asp-for="fullname"></label>
            <input asp-for="fullname" />
            <span asp-validation-for="fullname" style="color:red"></span>
        </div>
        <div>
            <label asp-for="roll"></label>

              <input asp-for="roll" value="" />
```

```
                <span asp-validation-for="roll" style="color:red"></span>
            </div>
            <div>
                <label asp-for="email"></label>
                <input asp-for="email" />
                <span asp-validation-for="email" style="color:red"></span>
            </div>
            <div>
                <input type="submit" value="Create"/>
            </div>
        </form>
    </body>
    </html>
```

3. **Now write the necessary action methods in the Home Controller one for displaying the above view for user input, and another to display the input values.**

```
using Microsoft.AspNetCore.Mvc;
using WebApplication59.Models;
namespace WebApplication59.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View(null);
        }
        [HttpPost]
        public IActionResult Submit(StudentViewModel s)
        {
            if (ModelState.IsValid) //For Server Side Validation
            {
                return Content("Full Name=" + s.fullname + " Roll=" + s.roll + " Email
                                                Address=" + s.email);
            }
            else
            {
                return Content("invalid model");
            }
        }
    }
}
```

- In the above example Model validation are performed at both side i.e client side and server side. We have used javascript libraries for client side validation and displaying the validation message summary. Similarly, when user click the submit button, we have again tested if the model is valid or not by testing ModelState.IsValid property. This validation test is performed at server hence called server side validation.
- It is good practice to perform validation both at client side and server side because we can't trust client validation as user can change the validation rules performed on the client. So validating on the server ensures everything is as we expect them to be. Since, validation on the server requires extra round-trip thereby making the validation process slower so it is better practice first to perform validation on client and if something is wrong no need server side validation. On the other hand if no errors on client side validation, we make call to the server and perform server side validation.

Custom Validation with Validation Attribute

- Custom validation is useful when the inbuilt validation attribute doesn't solve our validation problem.
- To define custom validation attribute and work it similar to that of built in attribute, we have to create a class and inherit it from the ValidationAttribute class.
- After that, we can simply override IsValid() method and write our own validation logic.
- In this example, we will create a validation attribute that let us define a Date type of field that only accepts a date earlier than today's date.

Steps:

1. Create Validation class code

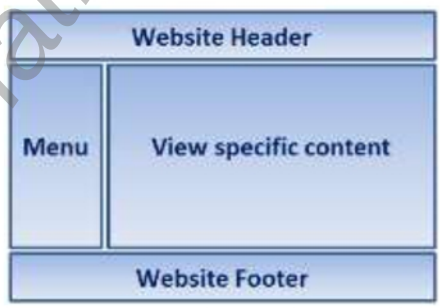
```
using System.ComponentModel.DataAnnotations;
public class DateValidationAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        DateTime todayDate = Convert.ToDateTime(value);
        if(todayDate<=DateTime.Now)
        {
            return (true);
        }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

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

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

Layout

- Most of the web applications in general have the following sections
  - Header
  - Footer
  - Navigation menu
  - View specific content



- Without a Layout View, we would repeat all the HTML for these different sections in every view in our application. With this duplicated HTML in every view, maintaining our application would be a night mare, if for example, we have to add or remove a menu item from the navigation menu or change header or footer. We would have to do this change, in every view, which is obviously tedious, monotonous, time consuming and error prone.
- Rather than having all of these sections, in each and every view, we can define them in a layout view and then inherit that look and feel in all the views. With layout views, maintaining the consistent look and feel across all the views becomes much easier, as we have only one layout file to modify, should there be any change. The change will then be immediately reflected across all the views in our entire application.
- Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

- Just like a regular view a layout view is also a file on the file system with a .cshtml extension
- We can think of a layout view as a master page in asp.net web forms.

- Since a layout view is not specific to a controller, we usually place the layout view in a sub folder called "Shared" in the "Views" folder
- By default, in ASP.NET Core MVC, the layout file is named _Layout.cshtml.

Example:

- The following is the default generated HTML in _Layout.cshtml.

```
<html>
<head>
   <title>@ViewBag.Title</title>
</head>
<body>
<p>This is Top Section</p>
   <div>
      @RenderBody()
   </div>
<p>This is Footer Section</p>
</body>
</html>
```

- Notice, the standard html, head, title and body elements are in this layout file. Since we now have them in the layout file, we do not have to repeat all this HTML in every view. View specific title is retrieved using @ViewBag.Title expression. For example, when "index.cshtml" view is rendered using this layout view, index.cshtml will set Title property on the ViewBag. This is then retrieved by the Layout view using the expression @ViewBag.Title and set as the value for the <title> tag.
- ViewBag does not provide intellisense and compile-time error checking. So using it to pass large amount of data from a regular razor view to a Layout view is not great, but for passing something very small like PageTitle, ViewBag is OK.
- @RenderBody() is the location where view specific content is injected. For example, if index.chtml view is rendered using this layout view, index.cshtml view content is injected at the location where we have @RenderBody() method call.

- Now, To render a view using the layout view (_Layout.cshtml) we need to set the Layout property. For example, to use the Layout view with index.cshtml, we have to include the Layout property as shown below.

```
@{
   Layout = "~/Views/Shared/_Layout.cshtml";
   ViewBag.Title = "HomePage";
   }
<div>
This is my homepage
</div>
```

- So the above view will be rendered on the location where @RenderBody() method is called in the layout.

Partial View

- A partial view is a reusable portion of a web page. It is .cshtml that contains HTML code.
- It can be used in one or more Views or Layout Views. You can use the same partial view at multiple places and eliminates the redundant code.
- Partial View encapsulates HTML and C# code that can be reused on multiple razor pages or view. Hence, partial view enables code reuse and reduce code duplication by managing reusable parts of views.
- For example, a partial view is useful for an author biography on a blog website that appears in several views. An author biography is ordinary view content and doesn't require code to execute in order to produce the content for the webpage. Author biography content is available to the view by model binding alone, so using a partial view for this type of content is ideal.
-

Example:

- It is not mandatory to create a partial view in a shared folder but a partial view is mostly used as a reusable component, it is a good practice to put it in the "shared" folder.
- Under shared folder create a partial view with following content and named it _MyPartialView.cshtml

```
<h1>I am From Partial View</h1>
```

- Now we can consume this partial view from any page using <partial> tag , html helper method as below

```
<html>
<body>
    <partial name="_MyPartialView" />
</body>
</html>

Or
<html>
<body>
@Html.Partial("_MyPartialView")
</body>
</html>
```

Passing Data to View

- We can pass data to views using several approaches:
  1. Strongly typed data: viewmodel
     - The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. We can pass an instance of the viewmodel type to the view from the action.
     - Using a viewmodel to pass data to a view allows the view to take advantage of *strong* type checking. *Strong typing* (or *strongly typed*) means that every variable and constant has an explicitly defined type (for example, string, int, or DateTime). The validity of types used in a view is checked at compile time.
     - Visual Studio and Visual Studio Code list strongly typed class members using a feature called **IntelliSense.** When you want to see the properties of a viewmodel, type the variable name for the viewmodel followed by a period (.). This helps you write code faster with fewer errors.
     - Specify a model using the @model directive.
     - We can use the model with **@Model:**

Example:

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

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

2. **Now write a strongly typed html view to display Student Detail by using above model.**
   View: Index.cshtml

```
@model WebApplication57.Models.StudentModel
<html>
<body>
    <div>
        Name:@Model.fullname
    </div>
    <div>
        Roll:@Model.roll
    </div>
    <div>
        Email Address:@Model.email
    </div>
</body>
</html>
```

3. **Now write the necessary action methods in the Home Controller one for displaying the above view with one student record.**

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

2. **Weakly typed data (View Data and View Bag)**
   - In addition to strongly typed views, views have access to a *weakly typed* (also called *loosely typed*) collection of data.
   - Unlike strong types, *weak types* (or *loose types*) means that you don't explicitly declare the type of data you're using.
   - You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.
   - For example: We can use collection of weekly typed data to pass list of items from controller to view for populating a dropdown list with a data.
   - This collection can be referenced through either the ViewData or ViewBag properties on controllers and views.
   - ViewData and ViewBag is used to pass the data from the controller action method to a view, layout or partial view and we can display this data on the view.
   - We can pass any type of data in ViewData and ViewBag like normal integer, string, even though you can pass objects. You can store any number of items as needed in the ViewData and ViewBag.
   - The ViewData and ViewBag is dynamically resolved at runtime, as a result, it does not provide any compiles time error checking as well as we do not get any intelligence. For example, if we miss-spell the key names then we wouldn't get any compile-time error. We get to know about the error only at runtime.
   - For that reason, some developers prefer to minimally or never use ViewData and ViewBag.
   - The data stored in the ViewData and ViewBag object exists only during the current request. As soon as the view is generated and sent to the client, the Viewdata and ViewBag object is cleared.

ViewData

- The ViewData in ASP.NET Core MVC is a dictionary of weakly typed objects which is derived from the ViewDataDictionary class.
- The ViewData property is a dictionary of weakly typed objects. ViewData uses the ViewDataDictionary type.
- The ViewData is work on the principle of Key-value pairs. This type of binding is known as loosely binding.
- How to Use?
  - First, we need to create a new key in ViewData and then assign some data to it. The key should be in string format and you can give any name to it and then you can assign any data to this key.

    ViewData[“KeyName”] = “Some Data”;
  - Since ViewData is a server-side code, hence to use it on view, we need to use the razor syntax i.e. @

    @ViewData[“KeyName”]
  - You can access the string data from the ViewData dictionary without casting the data to string type. But if you are accessing data other than the string type then you need to explicitly cast the data to the type you are expecting.

```
Student student = new Student()
{
    StudentId = 101,
    Name = "James",
    Branch = "CSE",
    Section = "A",
    Gender = "Male"
};

ViewData["Student"] = student;
```

Controller

```
@{
    var student = ViewData["Student"]
        as FirstCoreMVCApplication.Models.Student;
}
```

View

As the data stored in the ViewData dictionary is of Student Type so we need to explicitly cast it to Student Type

Example:

Controller Part

```
using Microsoft.AspNetCore.Mvc;
namespace WebApplication60.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            ViewData["Name"] = "Bheeshma";
            return View();
        }
    }
}
```

View Part

```
<html>
<body>
Student Name: @ViewData["Name"]
</body>
</html>
```

ViewBag

- ViewBag is a DynamicViewData object that provides dynamic access to the objects stored in ViewData.
- Unlike ViewData, ViewBag is more convenient to work with, since it doesn't require casting.
- The ViewBag uses the dynamic feature that was added in C# 4.0. It is a wrapper around the ViewData and provides the dynamic properties for the underlying ViewData collection.
- ViewBag is operating on the dynamic data type(var). So we don't require typecasting while accessing the data from a ViewBag. It does not matter whether the data that we are accessing is of type string or any complex type.

Storing the data in ViewBag

```
ViewBag.Header = "Student Details";
```

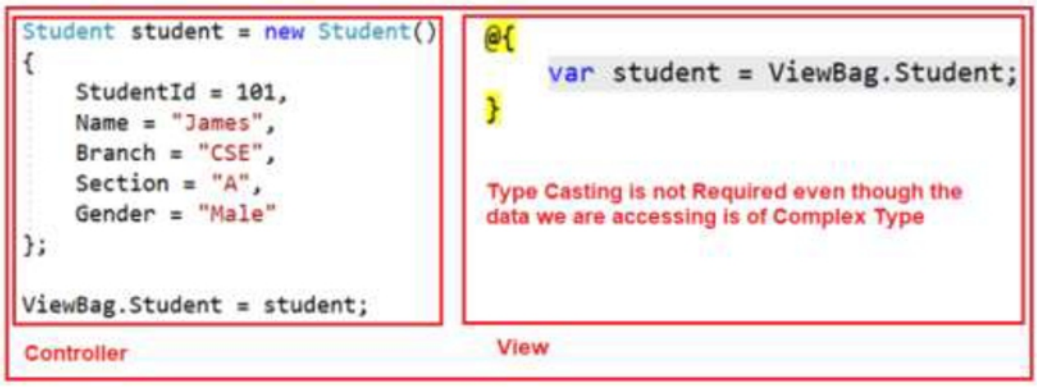
Controller

Accessing the Data

```
@ViewBag.Header
```

View

Type Casting is not Required



Example:
Controller Part

```
using Microsoft.AspNetCore.Mvc;
namespace WebApplication60.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            ViewBag.Name = "Bheeshma";
            return View();
        }
    }
}
```

View Part

```
<html>
<body>
Student Name: @ViewBag.Name
</body>
</html>
```

ViewBag vs ViewData

- The ViewData is a weakly typed dictionary object whereas the ViewBag is a dynamic property.
- In ViewData, we use string keys to store and retrieve the data from the ViewData dictionary whereas in ViewBag we use the dynamic properties to store and retrieve data.
- ViewData is derived from ViewDataDictionary, whereas ViewBag is derived from DynamicViewData, so it allows the creation of dynamic properties using dot notation
- Syntax for creating ViewData object is ViewData["key"]=<value> whereas Syntax for creating ViewBag object is ViewBag.key=<value> whereas
- Type casting is required in ViewDatta whereas typecasting not required in case of ViewBag

Dependency Inversion Principle (DIP)

- If function from class A class calls a function in class B, which calls a function in class C, then at compile time A will depend on B, which will depend on C, as shown in Figure below

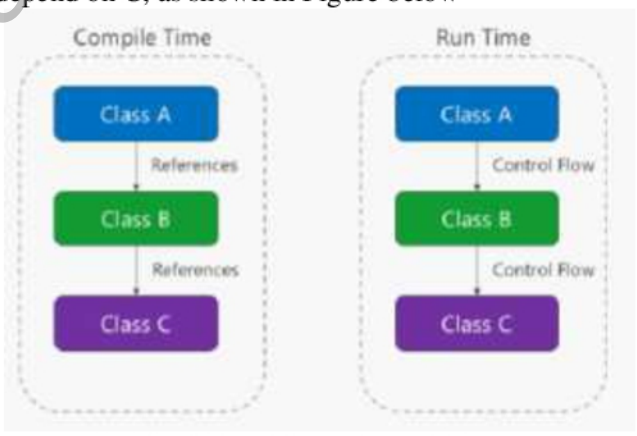


Fig: Direct Dependency Graph

- The principle behind **dependency inversion principle** is that
  1. High-level modules should not depend on low-level modules. Both should depend on the abstraction.

2. Abstractions should not depend on details. Details should depend on abstractions.

- Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at runtime. At run time, the flow of program execution remains unchanged, but the introduction of interfaces means that **different implementations** of these interfaces can easily be plugged in.

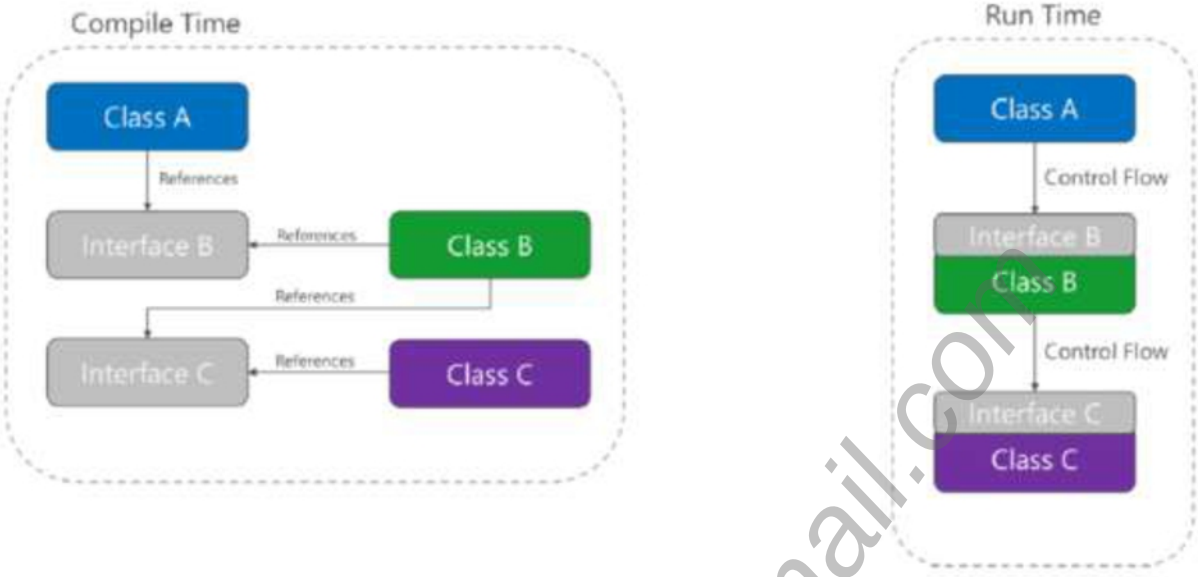
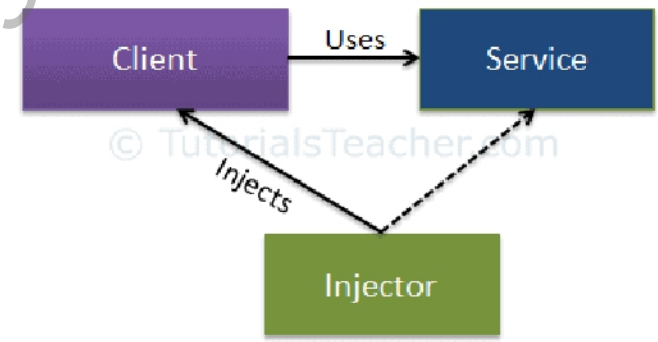


Fig: Inverted Dependency Graph

- Dependency inversion is a key part of building **loosely coupled applications**, since implementation details can be written to depend on and implement higher-level abstractions, rather than the other way around. The resulting applications are more testable, modular, and maintainable as a result.

Dependency Injection

- Dependency Injection (DI) is a design pattern used to implement DIP.
- In software engineering, dependency injection is a technique in which an object receives other objects that it depends on. These other objects are called **dependencies.**
- Generally the receiving object is called a client and the passed (that is, "injected") object is called a **service.**
- The code that passes the service to the client can be many kinds and is called the **injector.** Instead of the client specifying which service it will use, the injector tells the client what service to use. The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.
- Dependency injection involves four roles:
    1. the service object(s) to be used
    2. the client object that is depending on the service(s) it uses
    3. the interfaces that define how the client may use the services
    4. the injector, which is responsible for constructing the services and injecting them into the client



- A client who wants to call some services should not have to know how to construct those services. i.e. Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use Instead, the client delegates the responsibility of providing its services to external code (the injector). The client is not allowed to call the injector code; it is the injector that constructs the services. The injector then injects (passes) the services into the client which might already exist or may also be constructed by the injector. The client then uses the services. This means the client does not need to know about the injector, how to construct the services, or even which actual

services it is using. The client only needs to know about the interfaces of the services because these define how the client may use the services.

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

Example-1:

1. **Create Three Class named Employee, EmployeeDAL, EmployeeBL**

Model Class: Employee.cs

```
namespace DependencyInjectionExample
{
    public class Employee
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Department { get; set; }
    }
}
```

Data Access Layer: EmoloyeeDAL //Service

```
namespace DependencyInjectionExample
{
    public class EmployeeDAL
    {
        public List<Employee> SelectAllEmployees()
        {
            List<Employee> ListEmployees = new List<Employee>();
            //Get the Employees from the Database
            //for now we are hard coded the employees
            ListEmployees.Add(new Employee() { ID = 1, Name = "Pranaya", Department = "IT"
            });
            ListEmployees.Add(new Employee() { ID = 2, Name = "Kumar", Department = "HR"
            });
            ListEmployees.Add(new Employee() { ID = 3, Name = "Rout", Department =
            "Payroll" });
            return ListEmployees;
        }
    }
}
```

Business Logic Layer: EmployeeBl //Client

```
namespace DependencyInjectionExample
{
    public class EmployeeBL
    {
        public EmployeeDAL employeeDAL;
        public List<Employee> GetAllEmployees()
        {
            employeeDAL = new EmployeeDAL();
            return employeeDAL.SelectAllEmployees();
        }
    }
}
```

In the above example, in order to get the data, the **EmployeeBL** class **depends** on the **EmployeeDAL** class. This is **tight coupling** because the **EmployeeDAL** is tightly coupled with the **EmployeeBL** class. Every time the **EmployeeDAL** class changes, the **EmployeeBL** class also needs to change.

2. **We modify the EmployeeDAL as below**

```
namespace DependencyInjectionExample
{
        public interface IEmployeeDAL   //interfaces that defines services
        {
                List<Employee> SelectAllEmployees();
        }
        public class EmployeeDAL : IEmployeeDAL
        {
                public List<Employee> SelectAllEmployees()
                {
                        List<Employee> ListEmployees = new List<Employee>();
                        //Get the Employees from the Database
                        //for now we are hard coded the employees
                        ListEmployees.Add(new Employee() { ID = 1, Name = "Pranaya", Department = "IT"
                        });
                        ListEmployees.Add(new Employee() { ID = 2, Name = "Kumar", Department = "HR"
                        });
                        ListEmployees.Add(new  Employee()  {  ID  =  3,  Name  =  "Rout",  Department  =
                        "Payroll" });
                        return ListEmployees;
                }
        }
}
```

First we have created one interface i.e IEmployeeDAL with the one method. Then that interface is implemented by the EmployeeDAL class following the DIP principle i.e. a high-level module (EmployeeBL) and low-level module (EmployeeDAL) are dependent on an abstraction (IEmployeeDAL). Also, the abstraction (IEmployeeDAL) does not depend on details (EmployeeDAL), but the details depend on the abstraction.
Also, we can easily use another class which implements IEmployeeDAL with a different implementation easily.

3. **Modify the EmployeeBL as below**

```
namespace DependencyInjectionExample
{
        public class EmployeeBL
        {
                public IEmployeeDAL employeeDAL;
                public EmployeeBL(IEmployeeDAL employeeDAL)// Constructor Dependency Injection i.e.
                Injector
                {
                        this.employeeDAL = employeeDAL;
                }
                public List<Employee> GetAllEmployees()
                {
                        return employeeDAL.SelectAllEmployees();
                }
        }
}
```

In above code, we have created one constructor which accepts one parameter of the dependency object type. The parameter of the constructor is of the type interface, not the concrete class. Now, this parameter can accept any concrete class object which implements this interface.
So here in the EmployeeBL class, we are not creating the object of the EmployeeDAL class. Instead, we are passing it as a parameter to the constructor of the EmployeeBL class. **As we are injecting the dependency object through the constructor, it is called as constructor dependency injection in C#.**

4. **Finally, we will use EmployeeBL in main method of Program class as below**

```
namespace DependencyInjectionExample
{
        class Program
        {
                static void Main(string[] args)
                {
                        EmployeeBL employeeBL = new EmployeeBL(new EmployeeDAL());
                        List<Employee> ListEmployee = employeeBL.GetAllEmployees();
                        foreach(Employee emp in ListEmployee)
                        {
                                Console.WriteLine("ID = {0}, Name = {1}, Department = {2}", emp.ID,
                                emp.Name, emp.Department);
                        }
                        Console.ReadKey();
                }
        }
}
```

Output:

```
ID = 1, Name = Pranaya, Department = IT
ID = 2, Name = Kumar, Department = HR
ID = 3, Name = Rout, Department = Payroll
```

So we use one of the dependency injection techniques called constructor injection to make these classes loosely coupled.

Example-2: Logging Example

Program.cs

```
using System;
namespace ConsoleApp59
{
    class Program
    {
        static void Main(string[] args)
        {
            LoggingController l = new LoggingController(new DatebaseLogging());
            l.log("Program started");
            int a=5, b=6;
            l.log("Value Initialized");
            int c = a + b;
            l.log("Addion Completed");
            Console.Write("Sum=" + c);
            l.log("Program Terminated");
            Console.ReadKey();
        }
    }
}
```

LoggerController.cs

```
namespace ConsoleApp59
{
    class LoggingController
    {
        ILoggingInterface _ilog;
        public LoggingController(ILoggingInterface _ilog)//Injectors, constructor injection
        {
            this._ilog = _ilog;
        }
        public void log(string str)
        {
            _ilog.log(str);
        }
    }
}
```

IloggignInterface.cs

```
namespace ConsoleApp59
{
    public interface ILoggingInterface
    {
        void log(string str);
    }
}
```

DatabaseLogging.cs

```
using System;
namespace ConsoleApp59
{
    class DatebaseLogging : ILoggingInterface
    {
        public void log(string str)
        {
            Console.WriteLine("Database Logging" + str);
        }
    }
}
```

Filelogging.cs

```
using System;
namespace ConsoleApp59
{
    class FileLogging : ILoggingInterface
    {
        public void log(string str)
        {
            Console.WriteLine("Loggin to File "+str);
        }
    }
}
```


IoC Container

- ASP.NET Core is designed from scratch to support Dheshmathapa@gmail.com Injection.
- ASP.NET Core injects objects of dependency classes through constructor by default by using built-in IoC container.
- The followings are important interfaces and classes for built-in IoC container:
- The built-in container is represented by **IServiceProvider** implementation that supports constructor injection by default.
- The types (classes) managed by built-in IoC container is called **services**.
- In order to let the IoC container automatically inject our application services, we first need to register them with IoC container.

Registering Application Service

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

```
public interface ILog
{
      void info(string str);
}

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

- Now to register it with built-in IoC container and use it in our application, ASP.NET Core allows us to register our application services with IoC container, in the ConfigureServices method of the Startup class. The **ConfigureService**s method includes a parameter of **IServiceCollection** type which is used to register application services.
- Let's register above ILog with IoC container in ConfigureServices() method as shown below.

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

- As we can see above, Add() method of **IServiceCollection** instance is used to register a service with an IoC container. The **ServiceDescriptor** is used to specify a service type and its instance. We have specified ILog as **service type** and MyConsoleLogger as its instance. This will register ILog service with a **singleton lifetime** by default. Now, an IoC container will create a singleton object of MyConsoleLogger class and inject it in the constructor of classes wherever we include ILog as a constructor or method parameter throughout the application.
- Built-in IoC container manages the lifetime of a registered service type. It automatically disposes a service instance based on the specified lifetime.
- The built-in IoC container supports three kinds of lifetimes:
    - **Singleton:** IoC container will create and share a single instance of a service throughout the application's lifetime. Singleton lifetime services are created the first time they are requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance.
    - **Transient:** The IoC container will create a new instance of the specified service type every time you ask for it. Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.
    - **Scoped:** IoC container will create an instance of the specified service type once per request and will be shared in a single request. Scoped objects are the same within a request, but different across different requests.
- The following example shows how to register a service with different lifetimes.

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

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

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

- In addition to this, we can use other extension methods for quick and easy registration of services for each tyopes of lifetime. These are AddSingleton(), AddTransient() and AddScoped() methods for singleton, transient and scoped lifetime respectively.
- The following example shows the ways of registering types (service) using extension methods.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<ILog, MyConsoleLogger>();
    services.AddSingleton(typeof(ILog), typeof(MyConsoleLogger));

    services.AddTransient<ILog, MyConsoleLogger>();
    services.AddTransient(typeof(ILog), typeof(MyConsoleLogger));

    services.AddScoped<ILog, MyConsoleLogger>();
    services.AddScoped(typeof(ILog), typeof(MyConsoleLogger));
}
```

Constructor Injection

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

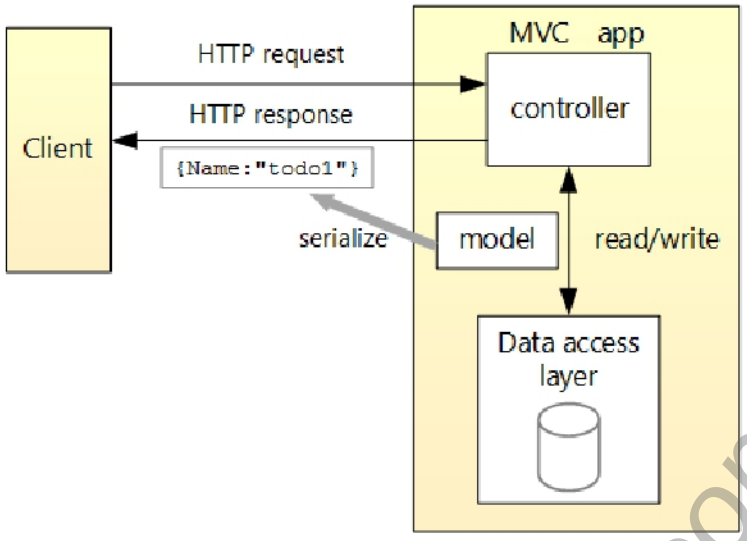
```
public class HomeController : Controller
{
      ILog _log;
    public HomeController(ILog log)//Constructor Injection
    {
      _log = log;
    }

    public IActionResult Index()
    {
      _log.info("Executing /home/index");
       return View();
    }
}
```

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

Web API Application

- A RESTful API is an architectural style for an application program interface (API) that uses HTTP requests to access and use data. That data can be used to GET, PUT, POST and DELETE data types, which refers to the reading, updating, creating and deleting of operations concerning resources.



- RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.
- ASP.NET Core supports creating JSON based RESTful services, also known as web APIs, using C#.
- To handle requests, a web API uses controllers class that is derived from ControllerBase class called as API Controller.
- Normally we don't create a web API controller by deriving from the Controller class as in MVC. This is because controller derives from Controller class adds support for views, so it's for handling web pages, not web API requests. There's an exception to this rule: if you plan to use the same controller for both views and web APIs, derive it from Controller.
- Web APIs can be accessible via simple JavaScript AJAX calls on any web page, whether it's a static HTML page or a View within an ASP .NET Core web app. We can also use your Web API as a backend for a mobile application running natively on a smartphone or tablet.
- The following program demonstrates the concept.

```
using Microsoft.AspNetCore.Mvc;
namespace WebApiDemo.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public ActionResult<string> Get(int id)
        {
            return "value";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }

        // PUT api/values/5
```

```
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}
```

MVC vs Web API

- Asp.Net MVC is used to create web applications that returns both views and data but Asp.Net Web API is used to create full blown HTTP services with easy and simple way that returns only data not view.
- In the Web API the request performs tracing with the actions depending on the HTTP services but the MVC request performs tracing with the action name.
- The Web API supports content negotiation(it's about deciding the best response format data that could be acceptable by the client. it could be JSON,XML or other formatted data. Content negotiation occurs when the client specifies an Accept header) but MVC doesn't.
- The Web API includes the various features of the MVC, such as routing, model binding but these features are different and are defined in the "System.Web.Http" assembly. And the MVC features are defined in the " System.Web.Mvc" assembly.
- Web API also takes care of returning data in particular format like JSON,XML or any other based upon the Accept header in the request and you don't worry about that. MVC only return data in JSON format using JsonResult.
- Web API is light weight architecture over MVC.