**ASP.NET**

**Steps for Updating**:

1. Create reference of Entity (Student) class that has to be updated by calling First or FirstOrDefault or SingleOrDefault or Single method on the table (Students).
2. Re-assign values to properties of the reference so that old values get changed to new values.
3. Call SubmitChanges method on DataContext object for saving the changes to Database server.

**Steps for Deleting**:

1. Create a reference of Entity (Student) class that must be deleted by calling First or FirstOrDefault or SingleOrDefault or Single method on the table (Students).
2. Call DeleteOnSubmit method on the table (Students) that deletes the record from table in a pending state.
3. Call SubmitChanges method on DataContext object for saving the changes to Database server.

**Eager Loading Vs Lazy Loading in Entity Framework:** eager loading is the process whereby a query for one type of entity also loads related entities as part of the query, so that we don’t need to execute a separate query for related entities. Eager loading is achieved using Include() method. In our above project under the “DisplayEmployees” action method we have accessed data of Employee entity as following:

MVCDBEntities dc = new MVCDBEntities();

var Emps = dc.Employees.Where(E => E.Status == true);

Default loading is lazy loading only and if we want to use eager loading then we need to first set “<DbContextClass>.Configuration.LazyLoadingEnabled = false”, and the above code should be replaced as below:

MVCDBEntities dc = new MVCDBEntities();

dc.Configuration.LazyLoadingEnabled = false;

var Emps = dc.Employees.Where(E => E.Status == true).Include(E => E.Department);

Lazy loading is delaying the loading of related data, until you specifically request for it. It is the opposite of eager loading. For example, the Employee entity contains the Department entity. In the lazy loading, the context first loads the Employee entity data from the Database, and then it will load the Department entity when we access the Department property.

MVCDBEntities dc = new MVCDBEntities();

List<Employee> Emps = dc.Employees.ToList(); //Loads Employees details only

Department Dept = Emps[0].Department; //Loads Department for particular Employee only

**To convert the above code to eager loading, re-write it as below:**

MVCDBEntities dc = new MVCDBEntities();

dc.Configuration.LazyLoadingEnabled = false;

List<Employee> Emps = dc.Employees.Include(E => E.Department).ToList();

Department Dept = Emp[0].Department;

ASP.NET is an Open-Source Web Framework, created by Microsoft, for building Modern Web Apps and Services with .NET using C# or VB.NET or F# languages. ASP.NET extends the .NET platform with tools and libraries specifically for building Web Apps.

.NET is Platform Independent i.e., applications that are developed by using .NET can run on multiple Platform’s and to run .NET Applications on a machine that machine should be 1st installed with a software i.e., .NET Runtime. Microsoft provided us 3 different .NET Runtimes that came over a period, like:

* .NET Framework Runtime
* .NET Core Runtime
* .NET Runtime

.NET Framework Runtime was launched in the year 2002 and this is available only for Windows Platforms. The first version of this Runtime is 1.0 and the last version is 4.8.

.NET Core Runtime was launched in the year 2016 and this is available for multiple platforms like Windows, Linux, and Mac. The first version of this Runtime is 1.0 and the last version is 3.1.

.NET Runtime is also same as .NET Core Runtime which was launched in the year 2020 and this is also available for multiple platforms like Windows, Linux, and Mac. This is a combination of .NET Framework and .NET Core and evolved as “One .NET”. The first version of .NET Runtime started with 5.0 and we call this as .NET 5 and the latest is .NET 6.

In .NET Framework we have been provided with ASP.NET Framework for building Web Applications and under this we have different options like:

ASP.NET Web Forms

ASP.NET MVC

ASP.NET Web API

In .NET CORE, .NET 5, and .NET 6 we have been provided with ASP.NET Core Framework for building Web Applications and under this also we have different options again, like:

ASP.NET Core Web App (Razor Pages)

ASP.NET Core Web App (Model-View-Controller)

ASP.NET Core Web API

**Note:** ASP.NET Core is the open-source version of ASP.NET, that can be developed and run-on Windows, Linux, macOS, and Docker. This was first released in 2016 and is a re-design of earlier Windows-Only versions of ASP.NET. Performance is a key focus of ASP.NET Core which is faster than other popular web frameworks as per independent [TechEmpower benchmarks](https://www.techempower.com/benchmarks/#hw=ph&test=plaintext). Like the rest of .NET, ASP.NET is also open source on GitHub which has over 100,000 contributions and 3,700 companies have already contributed.

**Applications what we use day to day are divided into different categories like:**

Desktop Applications

Web Applications

Mobile Applications

**Routing:** This is responsible for matching incoming HTTP Requests and then dispatching those requests to the applications executable Endpoints. Endpoints are the applications units of executable request-handling code which are defined in the application and configured when the app starts. The Endpoint matching process can extract values from the requests URL and provide those values for request processing. All ASP.NET Core templates include routing in their generated code. Routing is registered in the middleware pipeline of Startup class in ASP.NET Core 5.0 and Program class in ASP.NET Core 6.0.

**MVC Supports 2 different types of Routings:**

1. Conventional Routing
2. Attribute Routing

**Conventional Routing:** this is a pattern matching system for URL, which maps incoming requests to a particular Controller and Action method. In conventional routing we set all the routes in “RouteConfig.cs” file in “ASP.NET MVC” by calling the method “MapRoute”, whereas in “ASP.NET Core MVC” we do that by calling “MapControllerRoute” method in Startup class for ASP.NET Core 5.0 and Program class for ASP.NET Core 6.0.

**Attribute Routing:** this is a simple routing mechanism, compared to conventional routing. All the concepts are just like the conventional approach only but here, we define routes and attributes on a Controller or Action method. This was introduced from “ASP.NET MVC 5” and was available in “ASP.NET Core MVC” also. To use attribute routing in “ASP.NET MVC 5” we need to call the method “MapMvcAttributeRoutes” in “RouteConfig.cs” file whereas in case of “ASP.NET Core MVC” we need to call the method “MapControllers” in Startup class for ASP.NET 5.0 and Program class in ASP.NET Core 6.0.

|  |  |  |
| --- | --- | --- |
| **Route Type** | **Method to use in MVC 5** | **Methods to use in MVC Core** |
| Conventional: | MapRoute | MapControllerRoute |
| Attribute: | MapMvcAttributeRoutes | MapControllers |

**TCP/IP Protocols for the Web:** web browsers and servers use TCP/IP protocols to connect to the Internet. Common TCP/IP protocols are:

**HTTP:** Hyper Text Transfer Protocol takes care of the communication between a Web Server and a Web Browser. HTTP is used for sending requests from a Web Client (Browser) to a Web Server, returning Web Content (Web Pages) from the Server back to the Client.

**HTTPS:** Secure HTTP, takes care of secure communication between a Web Server and a Web Browser. HTTPS typically handles Credit Card transactions and other sensitive data.

**FTP:** File Transfer Protocol, takes care of transmission of files between Computers.

**SMTP:** Simple Mail Transfer Protocol, used to send or receive Emails between Senders and Receivers.

**Till now you might have created some Web Pages by using HTML, Java Script, and CSS, and are able to access those pages from your local machines by using their physical path or address in the browser; but how can we access those Web Pages from remote machines within a network?**

**Ans:** If we want to provide access to the Web Pages, we have developed to remote machines we need to take the help of a Server known as “Web Server”.

**What is a Server?**

**Ans:** It is software which works on 2 principles like request and response. There are so many Servers software’s in the industry, like DB Servers (Oracle, SQL Server, and My SQL etc.), Application Servers, Web Servers, etc.

**What is the need of a Web Server?**

**Ans:** this software is used for taking request (HTTP Request) from clients in the form of a “URL (Uniform Resource Locator)” and then sends Web Pages as response (HTTP Response).

**What Web Server software’s are available for us to consume?**

**Ans:** There are so many Webservers software’s that are available in the market like Apache Web Server from Apache, Nginx Web Server from NGINX, IIS Web Server from Microsoft, GWS Web Server from Google, LiteSpeed Web Server from LiteSpeed Technologies.

**Working with IIS Web Server:**

IIS stands for Internet Information Services which is formerly known as Internet Information Server.

IIS Web Server is a product of Microsoft and more compatible for our ASP.NET and ASP.NET Core Applications.

IIS is a part of Windows OS which is generally installed on our machines along with OS, and to verify whether it is installed on your machine or not, open any Browser and type in the URL => <http://localhost>.

**Note:** If IIS is not installed on our machine, we get the below error: “HTTP Error 404. The requested resource is not found.”

**Installing IIS on our machine if not installed:** Go to Control Panel => click on Programs and Features => In the window opened click on “Turn Windows features on or off”, which opens another window called “Windows Features” => in the new window opened select the CheckBox “Internet Information Services” and also select each and every Sub-Item (Checkbox’s) inside of it and click on the Ok button which will install IIS on your machine.

**How to access a Web Page using IIS?**

**Ans:** When we install IIS Web Server on our machine it will provide us an admin console for managing IIS and we call it as “IIS Manager”, which can be launched by searching for “inetmgr” in the window search. Once IIS Manager is opened, in LHS of the window we find “Connections Panel” and in that Panel we find our Computer Name as Server Name (Server is the name of my Web Server because the name of my computer is Server) and when we expand it, we find a node called as “Sites” and under that we find a website with the name “Default Web Site” which is created when IIS is installed on our machines.

**What is a Web Site?**

**Ans:** Web Site is a collection of Web Pages, and a Web Server is a collection of Web Sites, i.e., a Web Server can contain 1 or more Web Sites under it and by default when IIS is installed on a machine there will be 1 Website already created, with the name as “Default Web Site”.

**What is Routing?**

**Ans:** Routing enables you to use URL’s that do not have to map to specific files in a Web Site. Because URL does not have to map to a file, you can use URL’s that are descriptive of the user’s action and therefore easily understood by the users.

By using routing, we can define URL patterns that map to request-handler files, but that do not necessarily include the names of those files in URL. In addition, we can also include placeholders in a URL pattern so that variable data can be passed to the request handler without requiring a query string.

**Can we define multiple routes in RouteConfig class?**

**Ans:** Yes, we can define multiple routes in the RouteConfig class, so that MVC Framework evaluates each route in sequence. It starts with the first configured route, and if the incoming URL doesn’t satisfy the URL pattern of the route, then it will evaluate the second route and so on. To understand this open “RouteConfig.cs” file and add the below code above “default” route:

**What is Dependency Injection (DI) Design Pattern?**

**Ans:** Dependency Injection is a process of injecting the object/instance of a class into another class that depends on it. The Dependency Injection is the most used “Design Pattern” nowadays to remove the dependencies between the objects/instances and this allows us to develop loosely coupled software components.

**Dependency Injection Pattern involves 3 types of classes:**

* **Client Class:** The Client class is a class that depends on the Service class.
* **Service Class:** The Service class is the interface that provides services to the Client class.
* **Injector Class:** The Injector class injects the service class object into the Client class.

**Dependency Injection in ASP.NET Core:** ASP.NET Core Framework is designed from scratch to provide in-built support for Dependency Injection. ASP.NET Core Framework injects objects/instances of dependency classes through constructor or method or property by using a built-in IoC (Inversion of Control) container.

**What are the advantages of using ASP.NET Core Dependency Injection?**

**Ans:** ASP.NET Core Dependency Injection allows us to develop loosely coupled software components. Using ASP.NET Core Dependency Injection, it is very easy to swap with a different implementation of a component/class.

**What type of Services ASP.NET Core Dependency Injection provides to us?**

**Ans:** there are two types of services that ASP.NET Core Dependency Injection provides us, those are:

1. **Framework Services:** services that are a part of ASP.NET Core Framework such as “IApplicationBuilder”, “IConfiguration”, “IServiceCollection”, “ILoggerFactory”, “IWebHostEnvironment”, etc.
2. **Application Services:** the services (custom types) which we as a programmer create for our application.

**Note:** To let the “IoC” container automatically inject our application services, we first need to register them with the “IoC” container.

**How to register a Service with ASP.NET Core Dependency Injection Container?**

**Ans:** We register a service with ASP.NET Core Dependency Injection Container within the “ConfigureServices” method of the “Startup” class in ASP.NET Core 5.0 and “Program” class in ASP.NET Core 6.0. Before we register a Service with the Dependency Injection Container, it is important to understand the lifetime of a service, i.e., when a client class receives the dependency object/instance through dependency injection, whether the instance it receives is unique or not, depends on the lifetime of the service. Setting the lifetime of the dependency object/instance determines how many times the dependency object needs to be created and we are provided with 3 options to set the lifetime of the service, those are:

1. **Singleton:** in this case, the “IoC” container will create and share a single object/instance of a service object throughout the application’s lifetime.
2. **Scoped:** in this case, the “IoC” container will create an object/instance of the specified service type once per request and will be shared in a single request.
3. **Transient:** in this case, the “IoC” container will create a new object/instance of the specified service type every time you ask for it (Single Call).

**How to register a service with Dependency Injection Container?**

**Ans:** ASP.NET Core Framework provides 3 extension methods to register a service with the ASP.NET Core Dependency Injection Container and those methods will determine the lifetime of that service.

* AddSingleton<Interface, Class>()
* AddScoped<Interface, Class>()
* AddTransient<Interface, Class>()

**Note:** The built-in “IoC” container manages the lifetime of a registered service i.e., it automatically disposes of a service object/instance based on the specified lifetime.

**AddSingleton:** when we use this method to register a service, then it will create a singleton service which means a single instance of that service is created and shared among all the components of the application that require it. This instance is created when the first request comes to the service (single instance for all users).

**AddScoped:** when we use this method to register a service, then it will create a scoped service i.e., an instance of the service is created once per each HTTP Request and uses that instance in other calls of the same request (one instance for each request).

**AddTransient:** when we use this method to register a service, then it will create a transient service. It means a new instance of the specified service is created each time when it is requested, and they are never shared (one instance for each method call).

**When to use what?**

**Ans:** In real-time, we need to register components such as Application-Wide Configuration as Singleton. Database access classes like Entity Framework Contexts are recommended to be registered as Scoped so that the connection can be re-used. If you want to run anything in Parallel, then it is better to register the component as Transient.

**What are the different Types of Dependency Injections in .NET Core?**

**Ans:** Injector class can inject the dependency objects (service class) into a client class in 3 different ways, those are:

1. **Constructor Injection:** when the Injector injects the dependency object i.e., service class object/instance through the client class constructor, then it is called Constructor Injection.
2. **Property Injection:** when the Injector injects the dependency object i.e., service class object/instance through the public property of client class, then it is called as Property Injection or Setter Injection.
3. **Method Injection:** when the Injector injects the dependency object i.e., service class object/instance through a public method of the client class, then it is called as Method Injection.

**What is a Middleware?**

**Ans:** ASP.NET Core introduced a new concept called Middleware. A middleware is nothing but a class which is executed on every request in ASP.NET Core Application. There will be multiple Middleware’s in an ASP.NET Core Application. It can be either Framework provided Middleware added via NuGet or our own custom Middleware. We can set the order of Middleware execution in the request pipeline. Each Middleware adds or modifies Http Request and passes control to the next Middleware component.

At the beginning of the pipeline, we need to place the “Middleware’s” that are necessary for auxiliary tasks (like logging or authentication), and that don't consume a lot of memory and processing time.

**Authentication and Authorization:** Authentication is the process of obtaining identification credentials such as username (Id) and password from a user and validating those credentials against some authority. If the credentials are valid, the user that submitted the credentials is considered an authenticated user. Once the user has been authenticated, the authorization process determines whether that user has access to a resource or not.

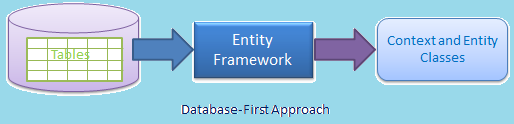
**ASP.NET supports different types of authentications like:**

* Anonymous Authentication
* Windows Authentication
* Forms Authentication
* Open Authentication

**Development Approaches with Entity Framework:** There are 3 different approaches you can use while developing your application using Entity Framework:

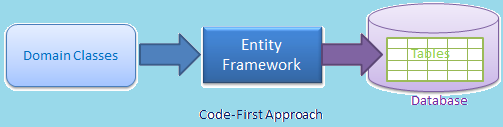
* **Database-First**
* **Code-First**
* **Model-First**

**Database-First Approach:** in this approach we generate the context and entity classes (model classes) for an existing database using EDM Wizard which is integrated into Visual Studio or executing EF commands.

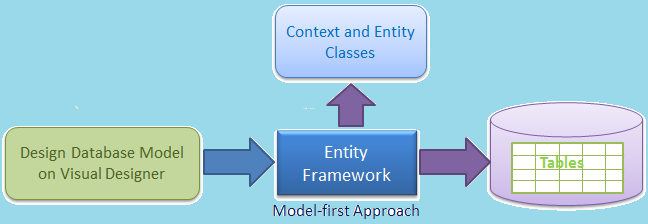


**Note:** EF 6 supports the database-first approach extensively whereas EF Core includes limited support for this approach.

**Code-First Approach:** in this approach we don’t have any existing database for our application and in such case, we start writing our entities (model) and context classes first and then create the database from these classes using migration commands. Developers who follow the “Domain-Driven Design (DDD)” principles, prefer to begin with coding their domain classes first and then generate the database required to persist their data.



**Model-First Approach:** In this approach, you create entities, relationships, and inheritance hierarchies directly on the visual designer integrated in Visual Studio and then generate entities, the context class, and the database script from your visual model.



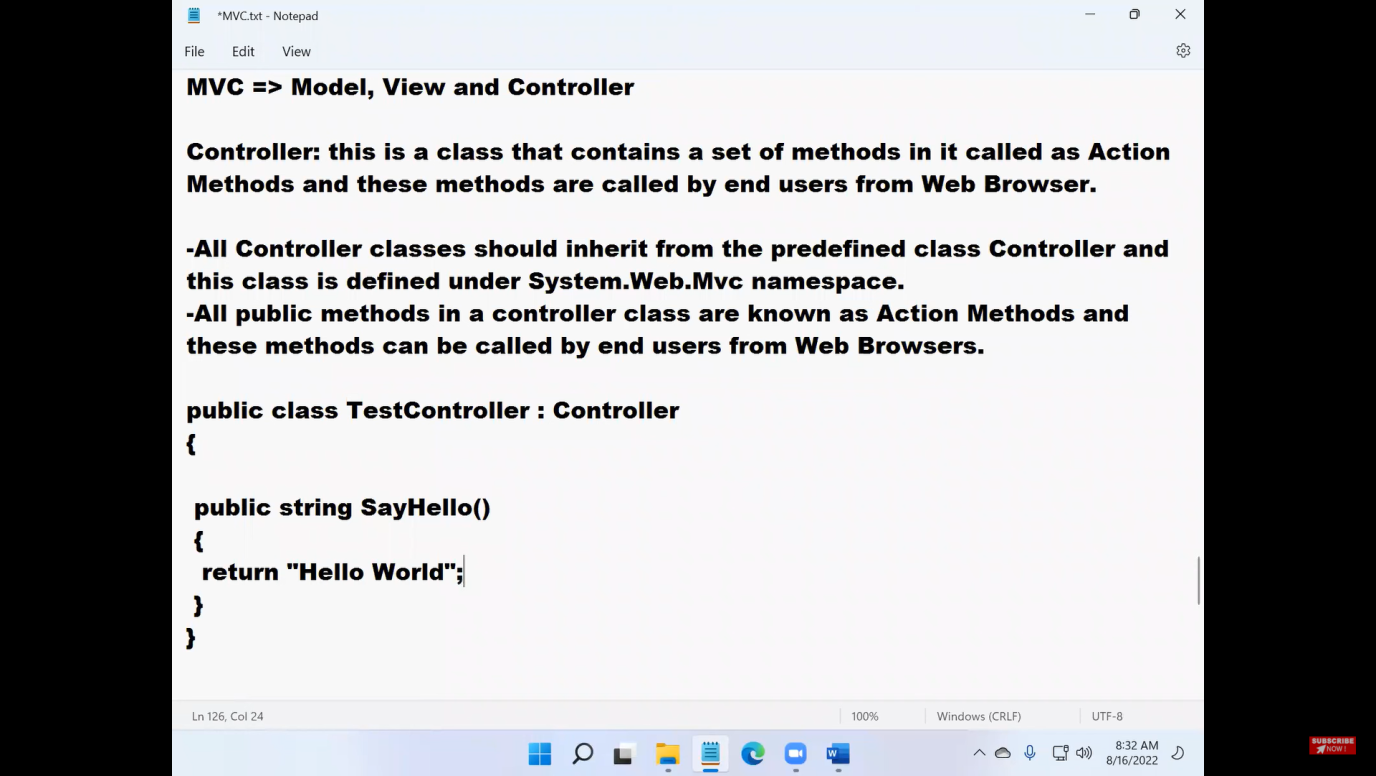
**Note:** EF 6 includes limited support for this, and EF Core does not support this approach.

**Architectural Patterns**

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. These are like software design pattern only but have a broader scope.

Software application architecture is the process of defining a structured solution that meets all the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability.

The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk and cost.



**What is MVC?**

**Ans:** The Model-View-Controller (MVC) is an architectural pattern which separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user and provides it with any Model data it requires. The following diagram shows the three main components and which ones reference the others:



**MODEL:** The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application.

**VIEW:** Views are responsible for presenting content through the user interface. There should be minimal logic within views, and any logic in them should relate to presenting content.

**Controller:** Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. 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 application responds to a given request).

**Controller**

It is a class that handles user requests i.e., this class is responsible for taking all the incoming requests for an MVC Application.

The parent class for all Controllers we define should be the class “Controller” which is in turn a child of class “ControllerBase” and both the classes are defined in “System.Web.MVC” namespace. Every Controller class should suffix the word “Controller” to it, for example if we want to define a controller with the name “Home” then it should be named as “HomeController”.

To test working with Controllers, create a new ASP.Net Web Application Project, naming it as “MVCTestProject3”, select Empty Project Template, check the MVC CheckBox, un-check all the other Checkbox’s and click on the Create button.

**Action Methods:** The methods that we defined under the Controller class for performing user interactions are known as Action methods i.e., users will directly call these methods for performing actions.

**To define Action Methods, we need to follow a set of rules:**

Action methods must be public, so every public method in a Controller class is an Action method only.

Action methods cannot be static because behind the screen instance of the Controller class is used for calling the Action methods.

It is not suggested to overload Action methods but if required we can still do that by decorating the method with “ActionName” attribute.

[ActionName("SayHello1")]

public string SayHello()

{

return "Hello how are you?";

}

[ActionName("SayHello2")]

public string SayHello(string Name)

{

return "Hello " + Name + " how are you?";

}

**Note:** in the above case we need to call the method with the “ActionName” we have defined by but not with the original method name, to test this define the above methods inside of “ParamsController” class and then we need to call them as following:

Action methods are generally value returning and very importantly in an MVC Application - Action Methods return type is an “ActionResult”, where “ActionResult” is a class and under this class there are a set of child classes and we call all those class as ActionResult only, and we can use any of those child classes as a return type of our Action method.

**List of ActionResult child classes are:**

ActionResult

FileResult

FilePathResult

FileStreamResult

FileContentResult

JsonResult

ViewResult

EmptyResult

ContentResult

RedirectResult

JavaScriptResult

PartialViewResult

HttpStatusCodeResult

RedirectToRouteResult

**General signature of an Action method will be as following:**

public <ActionResult> <Name>( [<Parameter List>] )

{

-Implement required logic here

-return an ActionResult

}

The most important ActionResult of an Action method is “ViewResult”, and a View in an MVC Application is the UI (User Interface) which contains all the presentation logic in it.

An Action method to return an “ActionResult”, we are provided with a set of methods known as “Helper Methods” and these helper methods are defined under Controller class, which is the parent or base class for all the controllers we define.

|  |  |
| --- | --- |
| **Helper Methods** | **Action Results** |
| File | FileResult |
| Json | JsonResult |
| View | ViewResult |
| --- | EmptyResult |
| Content | ContentResult |
| Redirect | RedirectResult |
| JavaScript | JavaScriptResult |
| PartialView | PartialViewResult |
| HttpNotFound | HttpStatusCodeResult |
| RedirectToRoute | RedirectToRouteResult |
| RedirectToAction | RedirectToRouteResult |

**Note:** all the above helper methods are defined under the class “Controller” and to see them go to “ParamsController.cs” file and in that file, right click on the “Controller” class and select the option “Go to Definition” which will take you to the pre-defined Controller class and display’s the metadata of that class.

**Views**

View is the second important component in an MVC Application which acts as a UI (User Interface) for presenting the data or results to end users as well as for accepting data from users. Views are stored under “Views” folder and under this folder a separate folder is maintained for storing the Views associated with each Controller i.e., if we have an “EmployeeController” then under the Views folder we will have “Employee” folder for storing all the Views that are associated with “EmployeeController” and so on.

**Note:** if a Controller is added by using Scaffolding, then automatically it will also add an associated folder for storing its Views, under the Views folder, whereas this will not happen if we define a Controller with manual coding and in that case, it is our responsibility to do that. Under the Views folder we can also maintain a folder with the name “Shared” for storing the Views that are common for multiple Controllers.

**What does a View contain in it?**

**Ans:** A View contains code for presentation or presentation logic which is a combination of “C# or VB” and HTML (CSS and Java Script also). When a request is sent for a View by the client, the logic implemented in the View gets processed and finally everything gets converted into Text (HTML) and we call this process as “Rendering”.

**What is rendering?**

**Ans:** Unfortunately, Internet still has bandwidth limitations and not every person is running on the same OS, same Web Browser or same Device, and these issues make it necessary to stick with HTML (Text Format) as our mark-up language of choice. So, in all the Server-Side technologies including ASP.NET; Web Server will process all the logic implemented by us using any language and converts the result into Text (HTML) which we call it as “**Rendering”** and then that HTML will be sent to clients as response. Views are processed by “View Engines” to render the results and MVC by default supports 2 different View Engines, those are:

Web Forms Engine

Razor Engine

**Web Forms Engine:** this is the default View Engine that is introduced along with MVC in 2008 and the coding style will exactly be like ASP.NET Web Forms, and the extension of View Pages here is “.aspx”.

**Razor Engine:** this is introduced in MVC 3.0 and in this case, View Pages will be having an extension of “.cshtml” or “.vbhtml” based on the language we use for developing the Views.

**Note:** Razor Engine is the most advanced View Engine and the most recommended also. Pages that are created for Razor Engine are known as Razor Pages and these Pages can contain either “HTML and C# or VB” code in them with an easy syntax.

**Razor Programming**

From MVC 3.0 Microsoft introduced Razor Engine for creating View Pages without using Web Form Pages. View Pages that are created targeting Razor Engine are saved with “.cshtml” extension whereas View Pages that are created targeting Web Form Engine are saved with “.aspx” extension.

Web Form Pages are provided with Design View, Source View and Code View, where we will be using Design View for design the UI with a “drag & drop” feature and Source View for implementing HTML, Java Script and CSS, and Code View for implementing C# Logic whereas Razor Pages is provided only with Source View and here only we can implement HTML, Java Script, CSS and C# Logic also, and this is the reason why these pages are saved with “.cshtml” extension.

Razor View Pages are light weight when compared to Web Form View Pages because we don’t use any ASP.NET Server Controls but will use only Html Controls so doesn’t require maintaining of View State.

Razor View Pages uses “@{ ... }” sign to implement C# code, whereas in Web Form Pages if we want to write any C# code it should be under “<% ... %>” tags.

**Code in Razor View Page can be written in 3 different ways:**

Single Line Statements.

In-Line Statements.

Multi Line Statements.

**Single Line Statements:** these are generally for declarations and Initializations.

**Syntax:**

@{ <Stmt>; }

**Examples:**

@{ int Count = 0; }

@{ Count += 100; }

@{ Object obj = new Object(); }

**In-Line Statements:** these statements are generally used for accessing values of Members, just by pre-fixing “@” character before the Member.

<h3>Value of count is: @Count</h3>

<h3>obj is of type: @obj.GetType() </h3>

**Multi Line Statements:** we use this for writing multiple lines of code.

**Syntax:**

@{

<Stmt's>

}

**Example:**

@{

string Date = DateTime.Now.ToShortDateString();

string Time = DateTime.Now.ToShortTimeString();

<h3>Today's Date is: @Date</h3>

<h3>Current Time is: @Time</h3>

}

**Note:** in multiline statement block we can write HTML Code directly without enclosing them in double quotes, whereas if we want to use any static text, we need to either prefix it with “@:” or put it under “<text></text>” tags.

@{

string Date = DateTime.Now.ToShortDateString();

string Time = DateTime.Now.ToShortTimeString();

@:Today's Date is: @Date

<text>Current Time is:</text> @Time

}

Comments in razor programming should be under “@\* Comment Text \*@” and in a multiline statement block we can also use our CSharp style of single line commenting i.e., “//”.

**Passing values from Controller Action Methods to Views in an MVC Application:** we are already aware that in an MVC Application all requests are handled by a Controller and these Controllers only will receive the information either from the End User or Model and these values should be sent to the View for displaying. To pass values from a Controller’s Action method to a View we are provided with various options like:

* ViewData
* ViewBag
* TempData
* Cookies
* Session
* Application
* Anonymous Types
* Model Objects

**ViewData:** this is a property defined under the class “ControllerBase” which is a grandparent for all our Controllers, so we can directly consume “ViewData” property in our Controller classes. **ViewData** is of type “ViewDataDictionary” which will internally store the data in the form of “Key-Value” or “Name-Value” combinations, so values that are stored in this “ViewData” will internally be stored in “ViewDataDictionary”.

**Storing values into ViewData:**

**Syntax:** ViewData[string Key] = Value (object)

**Example:** ViewData["Name"] = "Raju";

**Accessing values from ViewData:**

**Syntax:** Object obj = ViewData[string Key]

**Example:** Object obj = ViewData["Name"];

string Name = obj.ToString();

Or

string Name = ViewData["Name"].ToString();

**Drawbacks of ViewData:**

* ViewData can transfer data from a Controllers Action method to its corresponding View only.
* ViewData life lasts only during the current HTTP request i.e., ViewData values will be cleared if redirection occurs.
* ViewData is resolved dynamically at runtime, as a result, it doesn’t give any compile-time error checking as well as we do not get support of Intellisense. For example, if we miss-spell the “Key Names” then we will not get any compile-time or runtime error also, whereas we come to know about the problem at the runtime because the value is not displayed.
* ViewData values must be converted into an appropriate type (un-boxing) before using them because they are present in object format, as we have performed in-case of Price value to calculate 10% Tax.

**Note:** we can store scalar as well as complex types also in ViewData but while accessing them we need to explicitly convert them into its original type again. To test this, add a new Action method in “HomeController” class as below:

**ViewBag:** this is also a property defined in the class “ControllerBase”, but this of type “dynamic”. Dynamic is a new type introduced in C# 4.0, capable of storing any type of value and this is type safe because it represents the value in its exact type in runtime. ViewBag was introduced in MVC 3.0 which was just a wrapper around the ViewData but was type safe, so it doesn’t require any type conversion while consuming.

**Storing a value into ViewBag:**

**Syntax:** ViewBag.PropertyName = Value;

**Example:** ViewBag.Name = "Raju";

**Accessing a value from ViewBag:**

**Syntax:** <type> var = ViewBag.PropertyName;

**Example:** string Name = ViewBag.Name;

**TempData:** this is also a property under the class “ControllerBase”, but this is of type “TempDataDictionary”. Both “ViewDataDictionary” and “TempDataDictionary” are child classes of “IDictionary” interface which is designed for storing data in [key-value] or [name-value] combination.

The difference between TempData and ViewData is TempData can maintain the state of values between multiple requests i.e., it can pass values from 1 action method to another action method which are present in the same controller or another controller also.

**Note:** “TempData” is also not type safe like “ViewData” so here also we need to perform type conversion while working with data and the only difference between “ViewData” and “TempData” is “TempData” can maintain the state of values between multiple requests.

**Cookies:** A Cookie is a small piece of text that is used to store user-specific information and that information can be read by the Web Application whenever user visits the site. When a user requests for a Web Page, Web Server sends not just a page, but also a cookie containing the date and time. Cookies are stored in browser memory or a folder on the user’s hard disk and when the user requests for the Web Page again, browser looks for the Cookies associated with the Web Page and sends them to the Server. Browser stores cookies separately for each different site we visit.

**Cookies are or 2 types:**

1. In-Memory Cookies
2. Persistent Cookies

In-Memory cookies are stored in browser's memory so once the browser is closed, immediately all the cookies that are associated with that browser window will be destroyed, and by default every cookie is In-Memory only. Persistent Cookies are stored on Hard Disk of the client machines, so even after closing the browser window also they will be persisting and can be accessed next time we visit the site. To make a cookie as persistent we need to set “Expires” property of Cookie with a “DateTime” value.

**Drawbacks of Cookies:**

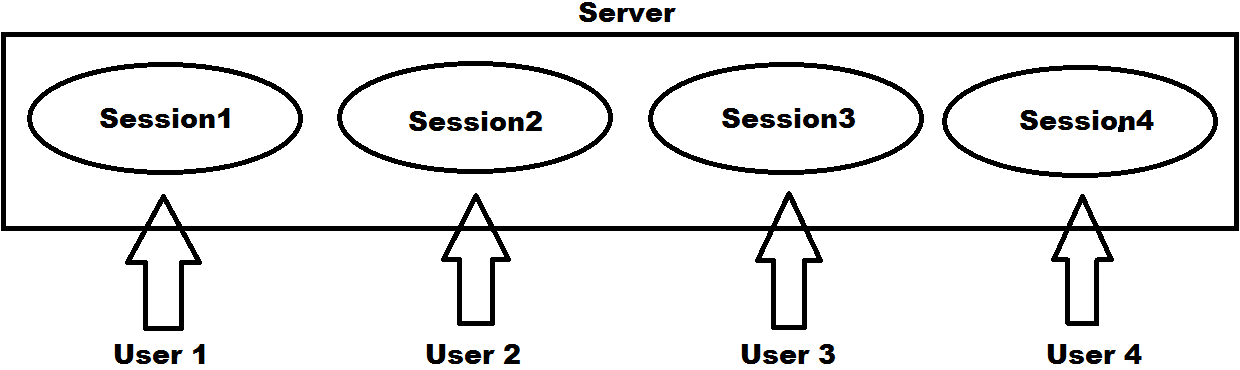
1. We can create only 50 cookies for each website, so every new cookie from the site will override the old cookie once after reaching the limit.
2. A cookie can store only 4 K.B. of data that too of type string only.
3. Cookies are not secured because they are stored on client machines.
4. Because cookies are stored on client machines there is a problem like clients can either delete the cookies or even disable cookies.

**Disabling cookies on Brower:** Click on “…” option beside the Address bar in browser => select settings and in the window opened, on the LHS select “Cookies and site permissions” and then on the RHS click on “Manage and delete cookies and site data” option and under that switch off the toggle button => “Allow sites to save and read cookie data (recommended)”.

**Delete Cookies on Browser:** To delete cookies on our browser use the command Ctrl + Shift + Delete which will open “Clear browsing data” window, in that select “Cookies and other site data” Checkbox and click “Clear now”.

**Session:** this is also a property but defined under “Controller” class and this is of type “HttpSessionStateBase”, and these are same as Sessions in “ASP.NET Web Forms”. Values that are stored in a Session are accessible from anywhere to anywhere for a particular user in the Session Life-time i.e., Action to Action in same Controller or other Controller, Controller to View, View to Controller, View to View etc.

Whenever a user connects to the Web Server, server will create a Session and gives it to the user for storing any values that are associated with him which are accessible only to him in every page he visits in the site. This happens for every user connecting to the server. So, each user will be having a Session of his own which will not be shared between other users, in short, a session is a “Single-User Global Data”.



Values that are stored in a Session will stay for any time with-out being deleted, but if at all a Session is in-active for a period of 20 Minutes (default, this can be changed), then the Session and its values will be destroyed.

**Note:** the default time-out of a Session is 20 Minutes i.e., if they are left idle for 20 Minutes then Session gets destroyed. We can change this time-out value to any new value in “Web.config” file of our project, specified in Minutes only and to test that open “Web.config” file and write the below statement under “<system.web>” tag:

<sessionState timeout="10" />

**Storing values into a Session:**

**Syntax:** Session[string key] = value (object)

**Example:** Session["Name"] = "Raju";

**Accessing values from a Session:**

**Syntax:** object value = Session[string key]

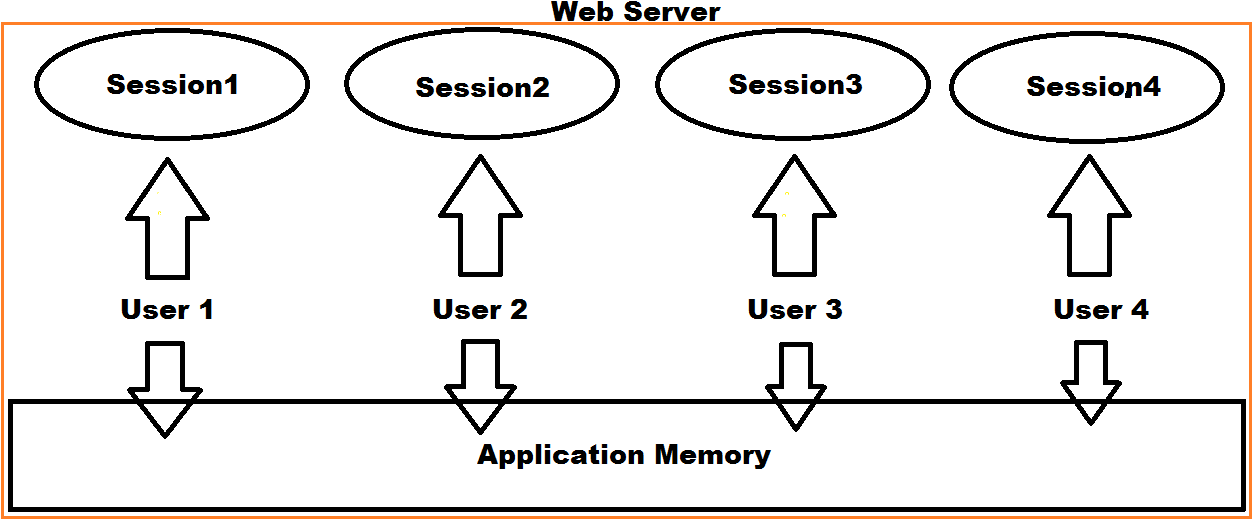
**Example:** object value = Session["Name"];

string Name = value.ToString();

Or

string Name = Session["Name"].ToString();

**Application:** this is a global storage mechanism that is used to store data on the server which is shared between all users i.e., data stored in Application is accessible to all users and anywhere in the application (Multi-User Global Data). Application-State is stored in the memory of the Web Server and is faster than storing and retrieving information from a Database. Application is used in the same way as Session, but Session is specific for a single user, whereas Application is common for all users of the application.



Application does not have any default expiration period like Session, so when we recycle the Worker Process or restart the Web Server only the data in application will be lost. Data is stored into Application in the form of name/value pairs only like we store in Session, ViewData or TempData. We store data into Application by using “Application” property of “HttpContext” class that is defined in “System.Web”.

**Note:** Application Memory is not Thread-Safe, so to overcome the problem whenever we are dealing with Application-State data we need to call Lock() and UnLock() methods on Application object.

**Storing values into Application:**

**Syntax:** System.Web.HttpContext.Current.Application[string key] = value (object)

**Example:** System.Web.HttpContext.Current.Application["Name"] = "Raju";

**Accessing values from Application:**

**Syntax:** object value = System.Web.HttpContext.Current.Application[string key]

**Example:** object value = System.Web.HttpContext.Current.Application["Name"];

string Name = value.ToString();

Or

string Name = System.Web.HttpContext.Current.Application["Name"].ToString();

**Anonymous Types:** This is another mechanism using which we can transfer values from Action method to View or Action method to Action method within the same Controller or another Controller also, without using ViewData, ViewBag, TempData and Session. An anonymous type is a type (class) without a name that contains a set of read only properties, for which we can directly create the instance by using “new” keyword.

**State Management:** Web Applications are stateless i.e., we can never access the values of 1 request, in the next request of the same page or other pages also. But sometimes we need the values of 1 request, in the next request to same page or other pages and to overcome this problem and maintain the state of values between multiple requests to the same page or between different pages we are provided with the concept called as State Management. In ASP.NET to maintain the state of values we are provided with various techniques like:

* Query String
* Hidden Field
* TempData
* Cookie
* Session
* Application

**UI Designing**

We design user interfaces in any application for taking input from end users or displaying the results. Designing of a user interface in Web Applications is performed with HTML, which provides with a set of controls. Whereas in ASP.NET MVC we call user interface as View, and we can design them by using any of the below options:

1. Using Html Controls
2. Using Html Helpers
3. Using Strongly Typed Html Helpers

**Methods of the “HtmlHelper” class:**

* BeginForm
* EndForm
* ActionLink
* CheckBox
* DropDownList
* Hidden
* ListBox
* Password
* RadioButton
* Label
* TextBox
* TextArea
* Display
* Editor

**Partial Views**

Just like we have “User Controls” in ASP.NET Web Forms we have “Partial Views” in ASP.NET MVC. A Partial View is also a View, but this can be used in other Views, i.e., they provide re-usability.

Creating a Partial View is also just like creating a normal View and the extension of the Partial View also will be “.cshtml” only, but in a Partial View we don’t have any HTML tag’s like “<head>” and “<body>” by default. Partial Views are generally stored in the “Shared” folder of “Views” folder because they are consumed by multiple Views in the application, and it is suggested (optional) to prefix Partial View names with “underscore (\_)” to differentiate them from other Views.

To work with Partial Views in our existing project i.e., “MVCUIDesigning” add a new folder with the name “Shared” in the “Views” folder and add another folder under the project with the name “Images”, copy an image into that folder and rename it as “Header”.

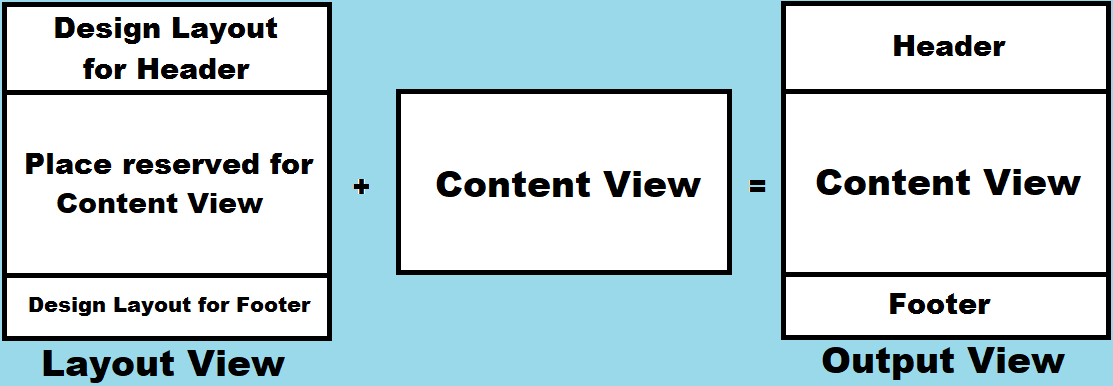
**Layout Views**

These are like “Master Pages” in “ASP.NET Web Forms” and by using these we can design a Layout for the whole application. Generally, every application will have a common Layout like the Header, Footer, Menu Bar, and Navigation Bar etc. So, in such scenario without re-designing them for multiple times in each View Page, we will design it as a Layout View and then use it in the whole Application.

In “ASP.NET Web Forms” the extension of a “Master Page” is “.master” whereas in “ASP.NET MVC” the extension of “Layout View” is also “.cshtml” only.

|  |  |  |
| --- | --- | --- |
|  | **ASP.Net Web Forms** | **ASP.NET MVC** |
| Page or View | .aspx | .cshtml |
| User Control or Partial View | .ascx | .cshtml |
| Master Page or Layout View | .master | .cshtml |

**Note:** Layout Views are also known as Master Pages and our View Pages are known as Content Pages or Content View so these Content Pages will merge with the Master Page to display the final output.

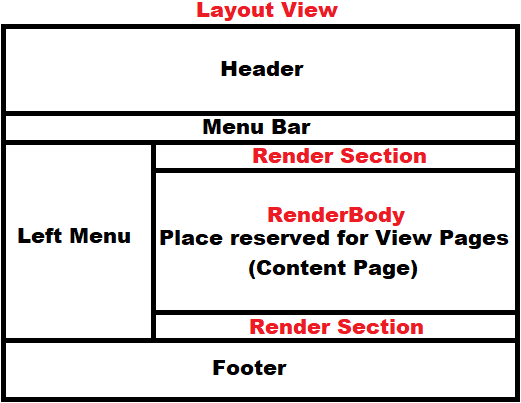


To include a Content View into a particular location of a Layout View we need to call a method i.e., “RenderBody” so that the Content View will come and sit in that location. A Layout View can contain 1 and only 1 “RenderBody” method call.

To include a “piece of text” into any location of a Layout View we need to call the method “RenderSection”, so that “text” can be sent into the Layout View from Content View and sending that “text” is not mandatory i.e., while calling the “RenderSection” method we can specify a name to the section and a “boolean” value to indicate, sending a value to that section is mandatory or optional.

RenderSection(string name, bool required)

**Note:** A Layout View can have any no. of “RenderSection” method calls in it, and in this process for every “RenderSection” we need to give a unique name for identification.



**Action Results in MVC**

Action Methods in an “MVC Application” can return different types of results and we call them as Action Results. In MVC, “ActionResult” is a class, and this class has various child classes under it and an Action method can return any of those classes as a result i.e., either parent class “ActionResult” or its child classes. Controller class (the parent class of all the Controllers we define) provides different helper methods to return a Result, where each helper method returns a different “ActionResult”, for example:

|  |  |
| --- | --- |
| **Helper Methods** | **Action Results** |
| File | FileResult |
| Json | JsonResult |
| View | ViewResult |
| --- | EmptyResult |
| Content | ContentResult |
| Redirect | RedirectResult |
| JavaScript | JavaScriptResult |
| PartialView | PartialViewResult |
| HttpNotFound | HttpNotFoundResult |
| HttpStatusCodeResult | HttpUnauthorizedResult |
| RedirectToRoute | RedirectToRouteResult |
| RedirectToAction | RedirectToRouteResult |

* **ViewResult:** sends a view as response.
* **PartialViewResult:** sends a partial view as response002E
* **RedirectToRouteResult:** represents a result that performs a redirection to an action method by using the specified route values dictionary.
* **JsonResult:** represents a class that is used to send JSON-Formatted content as response.
* **FileResult:** represents a class that is used to send binary file content as response.
* **RedirectResult:** controls the processing of application’s action method by redirecting to a specified URI.
* **ContentResult:** represents a user-defined content type, which is sent as a response.
* **JavaScriptResult:** sends JavaScript content as a response (currently this is not supported by the modern browsers and removed in MVC Core).
* **EmptyResult:** represents a result that does nothing, such as controller action method which returns void.
* **HttpStatusCodeResult:** provides a way to return an action result with a specific HTTP Status Code and Description.

**Minification and Bundling**

These are two techniques we use in Web Applications to improve request load time. Bundling and Minification improves load time by reducing the size of requested assets and reducing the number of requests to the server for the assets (such as CSS and Java Script.)

**Note:** most of the current major browsers limit the number of simultaneous connections per each host to 6. That means while 6 requests are being processed, additional requests for assets on a host will be queued by the browser.

**Minification:** this performs a variety of different code optimizations to CSS and Scripts such as removing unnecessary white spaces, removing comments, and shortening variable names to one character. For example, consider the below JavaScript function:

function SayHello(name) {

//This function wishes the user with his given name

var msg = "Hello " + name;

alert(msg);

}

**After Minification, the function is reduced as below:**

function SayHello(n){var m="Hello "+n;alert(m);}

In the above case “name” parameter is shortened as “n” and “msg” variable is shorted as “m”, removed comments, line breaks and unnecessary white spaces.

**Areas**

It is an approach of dividing a large “MVC Applications into smaller logical units, so that organizing of the application becomes much simple and easier. Area in an MVC Application is a collection of “Controllers”, “Views” and “Models” i.e., we can maintain them separately for each module.

For example, in a Hospital Management Application, for better development of the Application the Project Manager divided the Application into different modules like Patient Module, Doctor Module, Staff Module, Insurance Module, Billing Module, Labs Module, Medicine Module and HR Module. Now each Module is given for a different team to develop, so every Module will be having its own Controllers, Views and Models, for example:

Patients => Controllers, Views and Models

Doctors => Controllers, Views and Models

Staff => Controllers, Views and Models

Insurance => Controllers, Views and Models

Billing => Controllers, Views and Models

Labs => Controllers, Views and Models

Medicines => Controllers, Views and Models

HR => Controllers, Views and Models

While integrating all these Modules, for a clear separation and management of the Application we use areas because every area will be having its own Controllers, Views and Models, so under the project we maintain 1 area for each Module as below:

PatientArea => PatientControllers, PatientViews, PatientModels

DoctorArea => DoctorControllers, DoctorViews, DoctorModels

StaffArea => StaffControllers, StaffViews, StaffModels

InsuranceArea => InsuranceControllers, InsuranceViews, InsuranceModels

BillingArea => BillingControllers, BillingViews, BillingModels

LabArea => LabControllers, LabViews, LabModels

MedicinesArea => MedicineControllers, MedicineViews, MedicineModels

HRArea => HRControllers, HRViews, HRModels

**Data Annotations**

These are nothing but validations that we put in our “Models” to validate the input from the user. These are similar to “Validation Controls” in “ASP.NET Web Forms”. “ASP.NET MVC” provides a unique feature by using which we can validate the models by using the “Data Annotation” attributes importing the namespace “System.ComponentModel.DataAnnotations”. Data Annotations can be used in our View Pages for validating Controls as well as they can also be used in “Entity Framework”. Data Annotations allows us to define all the rules a Model Class and its Properties has to follow, and they are divided into different categories, like:

* Display Attributes
* Validation Attributes
* Modeling Attributes

Every Data Annotation is a class that is defined in the libraries of our language under the namespace “System.ComponentModel.DataAnnotations” and the parent class for all these classes is “Attribute” and the hierarchy of classes is as following:

* Attribute
  + DisplayAttribute
  + DisplayFormatAttribute
  + DisplayColumnAttribute
  + ValidationAttribute
    - RequiredAttribute
    - DataTypeAttribute
    - CompareAttribute
    - RangeAttribute
    - RegularExpressionAttribute
    - RemoteAttribute (This class is defined under “System.Web.MVC” namespace)

**Note:** Data Annotation’s - Validation Attributes, will perform data validations both on Client as well as Server also i.e., first they will validate the data on Client Machine and if at all those validations fail, page will not be submitted to the Server, whereas if the Client disables Java Script on his browser, then data is submitted to Server even if the Validations fail and to overcome this problem validations are re-performed on Server also.

To perform Client-Side Validations, Data Annotations uses JQuery Library, so our View Pages have to use the below JQuery libraries:

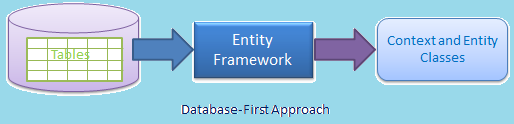
* jquery-<version>.min.js
* jquery.validate.min.js
* jquery.validate.unobtrusive.min.js

**Entity Framework**

**Development Approaches with Entity Framework:** There are 3 different approaches you can use while developing your application using Entity Framework:

* **Database-First**
* **Code-First**
* **Model-First**

**Database-First Approach:** in this approach we generate the context and entity classes (model classes) for an existing database using EDM Wizard which is integrated into Visual Studio or executing EF commands.



**Note:** EF 6 supports the database-first approach extensively whereas EF Core includes limited support for this approach.

**Eager Loading Vs Lazy Loading in Entity Framework:** eager loading is the process whereby a query for one type of entity also loads related entities as part of the query, so that we don’t need to execute a separate query for related entities. Eager loading is achieved using Include() method. In our above project under the “DisplayEmployees” action method we have accessed data of Employee entity as following:

MVCDBEntities dc = new MVCDBEntities();

var Emps = dc.Employees.Where(E => E.Status == true);

Default loading is lazy loading only and if we want to use eager loading then we need to first set “<DbContextClass>.Configuration.LazyLoadingEnabled = false”, and the above code should be replaced as below:

MVCDBEntities dc = new MVCDBEntities();

dc.Configuration.LazyLoadingEnabled = false;

var Emps = dc.Employees.Where(E => E.Status == true).Include(E => E.Department);

To enable eager loading in “DisplayEmployees” and “DisplayEmployee” methods re-write the code under both the methods as below:

public ViewResult DisplayEmployees()

{

dc.Configuration.LazyLoadingEnabled = false;

var Emps = dc.Employees.Where(E => E.Status == true).Include(E => E.Department);

return View(Emps);

}

public ViewResult DisplayEmployee(int Eid)

{

dc.Configuration.LazyLoadingEnabled = false;

var Emp = (dc.Employees.Where(E => E.Eid == Eid).Include(E => E.Department)).Single();

return View(Emp);

}

Lazy loading is delaying the loading of related data, until you specifically request for it. It is the opposite of eager loading. For example, the Employee entity contains the Department entity. In the lazy loading, the context first loads the Employee entity data from the Database, and then it will load the Department entity when we access the Department property.

MVCDBEntities dc = new MVCDBEntities();

List<Employee> Emps = dc.Employees.ToList(); //Loads Employees details only

Department Dept = Emps[0].Department; //Loads Department for particular Employee only

**To convert the above code to eager loading, re-write it as below:**

MVCDBEntities dc = new MVCDBEntities();

dc.Configuration.LazyLoadingEnabled = false;

List<Employee> Emps = dc.Employees.Include(E => E.Department).ToList();

Department Dept = Emp[0].Department;

**MVC Filters**

In ASP.NET MVC, a user request is routed to an appropriate Controller and Action Method. However, there may be circumstances where you want to execute some logic before or after an Action Method executes or before or after an ActionResult executes, and to achieve this ASP.NET MVC provides Filters. Filters are used to inject extra logic at the different levels of MVC Framework’s request processing pipeline which provide a way for cross cutting concerns like logging, authorization and caching.

Initially ASP.NET MVC Framework supports four different types of filters like Authorization Filters, Action Filters, Result Filters and Exception Filters whereas Authentication Filters are introduced with ASP.NET MVC 5 and each Filter allows us to introduce logic at different points during the request processing pipeline.

In MVC, filters are classes, and all those classes implement some interface. The following table lists filter types and the interface which they implemented:

|  |  |  |
| --- | --- | --- |
| **Filter Types** | **Interface** | **Description** |
| Authentication | IAuthenticationFilter | These filters run before any other filters run or action method executes. |
| Authorization | IAuthorizationFilter | These filters run next of Authentication Filters and before any other filters run or action method executes. |
| Action | IActionFilter | These filters run before and after an action method executes. |
| Result | IResultFilter | These filters run before and after the action result executes. |
| Exception | IExceptionFilter | These filters run only if any filter or action method or action result throws an exception. |

**Authentication Filters:** Authentication filter runs before any other filter or action method executes. Authentication confirms that you are a valid or invalid user, and these filters must implement the IAuthenticationFilter interface.

**Authorization Filters:** Authorization Filters are responsible for checking User Access and these filters must implement the IAuthorizationFilter interface. These filters are used to implement authorization for controllers and action methods. The Authorize Attribute is the example of Authorization Filters.

**Action Filters:** Action Filters can be applied to a controller action method or an entire controller. These filters will be called before the action method starts executing and after the action method has executed. Action filters implement the IActionFilter interface that has 2 methods OnActionExecuting and OnActionExecuted. OnActionExecuting method executes before an action method and gives an opportunity to cancel the Action call and OnActionExecuted method executes after an action method and gives an opportunity to modify the view data that a controller action method returns.

**Result Filters:** These filters contain logic that is executed before and after a view result is executed like if we want to modify a view result right before the view is rendered to browser. OutputCacheAttribute is an example of Result Filter and these filters implement IResultFilter interface which contains OnResultExecuting and OnResultExecuted methods.

**ExceptionFilters:** These filters can be used to handle errors raised by either a controller action methods or action results i.e., they execute if there are any unhandled exceptions thrown during the execution pipeline. The HandleErrorAttribute is an example of ExceptionFilters, and they implement IExceptionFilter interface.

**List of pre-defined filters provided by ASP.NET MVC:**

**ChildActionOnly:** This filter ensures that an action method can be called only as a child method from a view of another action method. We tend to use this filter to prevent action methods from being invoked directly and if we try to do that it will throw an error.

**OutputCache:** One of the best ways to improve the performance of an ASP.NET MVC Application is by caching. With the help of caching, we can reduce hosting server and database Server round trips. We can apply OutputCache Action Filter to achieve caching either on an Action Method or on the whole controller. OutputCache filter has several properties like CacheProfile, Duration, Location, VaryByParam, VaryByHeader etc.

* **Duration:** Gets or sets the cache duration in seconds.
* **VaryByParam:** Gets or sets the vary-by-param value and if not specified default value is none.
* **VaryByCustom:** Gets or sets the vary-by-custom value.
* **Location:** Gets or sets the location value which is to specify where the output must be cached, it takes an Enum value which can be Server, Client, Downstream (Proxy Server), ServerAndClient, Any and None, default is any.
* **CacheProfile:** Gets or Sets the cache profile value from Web.config file.

**ValidateInput:** Cross Site Scripting (CSS/XSS) attack is very common and is a well-known attack for web applications, for example a CSS/XSS attack is basically the result of poor form validation. How CSS/XSS attacks work is at first the hacker does inject some HTML code into a HTML input field and the data along with the HTML tag is saved to the database. Now, when there is a need to display the data in a user interface then we will get it from the Database and a legitimate browser will parse it as HTML code. If the hacker then injects a normal HTML string, then there is no problem at all but if they inject harmful Java Script code from an input field that might steal valuable information from the user’s computer, but we are very sure that we never want to allow a user to inject a HTML element through a form. In traditional Web Form applications, we use a form validation script (in Java Script very often) to validate user’s input whereas in MVC the library has done all the job for us, so we need not validate or write lengthy code externally. In MVC by default it prevents the HTML element as form data, anyway we can use the ValidateInput attribute to prevent HTML explicitly in this way: [ValidateInput(true)] which can be used either on controller or action method.

**ValidateAntiForgeryToken:** This is a built-in functionality provided by Microsoft which developers often use in their applications for security purposes i.e., to stop CSRF (Cross Site Request Forgery) from hackers. Cross Site Request forgery can be defined as, a forgery request, i.e., a fraud or fake request, which comes on an authenticated site from a cross site and is treated as an authenticated request. For avoiding this situation, Microsoft provides ValidateAntiForgeryToken functionality which we can use in our application so that no one can hack our site or invade some critical information.

**HandleError:** This is an Exception Filters to handle errors in ASP.NET MVC which can be applied over the action method as well as Controller or at the global level. The HandleError Error filter has a set of properties that are very useful in handling the exception. This comes under Exception Filter.

* **ExceptionType:** Type of exception to be catch. If this property is not specified, then the HandleError filter handles all exceptions.
* **View:** Name of the view page for displaying the exception information.
* **Master:** Master View for displaying the exception.
* **Order:** Order in which the action filters are executed. The Order property has an integer value, and it specifies the priority from 1 to any positive integer value. 1 means highest priority and greater the value of the integer is, the lower is the priority of the filter.
* **AllowMultiple:** indicates whether more than one instance of error filter attribute can be specified.

**Authorize:** specifies that access to a controller or action method is restricted to users who meet the authorization requirement. You can apply the Authorize attribute to individual methods as well as the controller class. If you add the Authorize attribute to the controller class, then any action methods on the controller will be available only to authenticated users. The Authorize attribute is inheritable which means that you can add it to a base controller class of yours and thereby ensure that any methods of the derived controllers are also subject to authentication.

**AllowAnonymous:** when applied to a method, the AllowAnonymous attribute instructs the ASP.MVC runtime to accept and process the call even if the caller is not authenticated. The scenario when the AllowAnonymous comes handy is when you apply Authorize at the class level and then need to enable free access to some methods like login action method, registration action methods etc. This comes under Authorization Filter.

**Custom Filters:**

In MVC, filters are used to inject logic at different levels of request processing and allow us to share logics across Controllers and Action Methods. We are provided with various filters in MVC which we have used above and apart from that we can also create our own filters to implement logic at various levels and we call them as “Custom Filters”. We write custom filters for various reasons, like logging or for saving data to a Database before any action execution or we could also create a filter for fetching data from the Database and setting them as global values of our application. For example, let’s say we want to run security logic or a logging logic across the controllers, and to do so; we can write a filter containing those logics and enable them across all controllers. When we enable a filter across all controllers or actions, the filter executes on all upcoming HTTP requests. As discussed earlier, in ASP.NET MVC we have 5 types of filters and the sequence of running those filters is as follows:

* The Authentication filter runs before any other filter or action method.
* The Authorization filter runs after the Authentication filter and before any other filter or action method.
* The Action filter runs before and after any action method.
* The Result filter runs before and after execution of any action result.
* The Exception filter runs only if filters or action methods or action results throw an exception.

**The below diagram, explains the sequence of filter execution as shown below:**



**Ajax and JQuery**

Ajax stands for Asynchronous JavaScript and XML is a web development technique using many web technologies on the client side to create asynchronous web applications. With Ajax, Web Pages can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By de-coupling the data interchange layer from the presentation layer, Ajax allows web pages to change content dynamically without the need to reload the entire page.

Ajax is not a single technology but rather a group of technologies like HTML and CSS are used in combination, to mark up and style the information. The Web Page will be modified by Java Script to dynamically display and allow the user to interact with the new information. The built-in “XMLHttpRequest” object within Java Script is used to execute Ajax on Web Pages allowing Web Applications to load content on to the screen without refreshing the page. XML or JSON are used for inter-change of the data. Ajax is not a new technology, or different language, just existing technologies used in new ways.

**Note:** different browsers implement the Ajax differently that means if you’re adopting the typical Java Script way to implement the Ajax you must write different code for different browsers to ensure that Ajax would work cross-browser. But fortunately, JQuery simplifies the process of implementing Ajax by taking care of those browser differences. It offers simple methods such as load(), $.get(), $.post(), etc. to implement the Ajax that works seamlessly across all the browsers. XML is commonly used as the format for receiving server data, although any format, including plain text, can be used. In practice, modern implementations commonly utilize JSON instead of XML due to the advantage of JSON being native to Java Script.

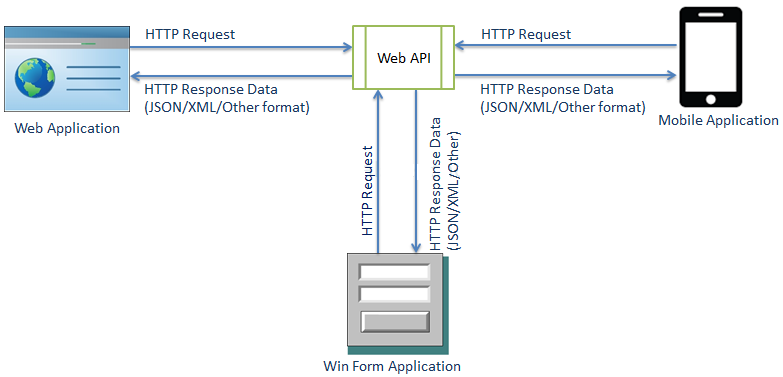
**ASP.NET Web API**

ASP.NET Web API (Application Programming Interface) is a framework for building HTTP Services that can be accessed from any client including browsers, desktops, and mobile devices. It is an ideal platform for building “Restful Applications” on the .NET Framework.

In computer programming, an API is a set of sub-routines (methods) definitions, protocols, and tools for building software and applications. Putting in simple terms, API is interface which has a set of methods that allow programmers to access specific features or data of an application, operating system, or other services.

Web API as the name suggests, is an API over the Web which can be accessed using HTTP protocol. It is a concept and not a technology. We can build Web API using different technologies such as Java, .NET etc. For example, Twitter’s REST APIs provide programmatic access to read and write data using which we can integrate twitter's capabilities into our own application.

The ASP.NET Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms such as Web, Windows, Mobile etc. It works the same way as an ASP.NET MVC Web Application except that it sends Data as a response instead of Views. It is like a Web Service or WCF Service, but the exception is that it only supports HTTP Protocol.



**Background of Web API:** Connectivity between applications is a very important aspect from a business applications perspective. Nowadays there are a lot of mobile applications and single page applications are being created and such applications needs a strong service end that can provide these applications with the data and CRUD operations on the data.

**SOAP and ASP.NET Web Services:** Traditionally, Web Services using SOAP and XML provided a great way of creating connected web applications. SOAP is a standard XML based protocol that communicates over HTTP. We can think of SOAP as message format for sending messages between applications using XML. It is independent of technology, platform, and language too. ASP.NET Web Services provided an excellent way of creating SOAP based Web Services.

**Problems with SOAP and Web Services:** SOAP offered an excellent way of transferring the data between the applications, but the problem with SOAP was that along with data a lot of other Metadata also needs to get transferred with each request and response. This extra information is needed to find out the capabilities of the service and other Meta data related to the data that is being transferred coming from the server. This makes the payload heavy even for small data. Also, Web services needed the clients to create the proxy on their end. These proxies will do the marshaling and un-marshaling of SOAP WSDL and make the communication between the application and the web service possible. The problem with this proxy is that if the service is updated and the proxy on the client is not, then the application might behave incorrectly.

**Introduction of REST:** REST stands for Representational State Transfer. This is a protocol for exchanging data over a distributed environment. The main idea behind REST is that we should treat our services as resources, and we should be able to use simple HTTP protocols to perform various operations on that resource. For example, when we talk about the Database as a resource, we usually talk in terms of CRUD operations i.e., Create (Insert), Retrieve (Select), Update and Delete. Now the philosophy of REST is that for a remote resource all these operations should be possible, and they should be possible using simple HTTP Protocols. Now the basic CRUD operations are mapped to the HTTP Protocols in the following manner:

GET: this maps to the R (Retrieve) part of the CRUD operation. This will be used to retrieve the required data from the remote resource.

POST: This map to the C (Create) part of the CRUD operation. This will create a new entry for the current data that is being sent to the server.

PUT: This map to the U (Update) part of the CRUD operation. This protocol will update the current representation of the data on the remote server.

DELETE: This map to the D (Delete) part of the CRUD operation. This will delete the specified data from the remote server.

Let’s take a simple example of some remote resource that contains a Database for list of movies and the list of movies can be retrieved using a URL like: [www.moviewebsite.com/api/movies](http://www.moviewebsite.com/api/movies)

To retrieve any specific movie, let’s think there is some ID for each movie that we can use to retrieve the movie; the possible URL might look like: [www.moviewebsite.com/api/movies/123](http://www.moviewebsite.com/api/movies/123)

Since these are GET requests, data can only be retrieved from the server. To perform other operations, if we use similar URI structure with PUT, POST or DELETE operation, we should be able to create, update and delete the resource from server. Now if we compare REST API with SOAP, we can understand the benefits of REST like:

* Firstly, only the data will be traveling to-and-fro from the server because the capabilities of the service are mapped to the URIs and Protocols.
* Secondly, there is no need to have a Proxy at the client end because its only data that is coming and the application can directly receive and process the data.

**WCF REST Services:** Windows Communication Foundation (WCF) came into existence much later than the web service. It provided a much secure and mature way of creating the services to achieve whatever we were not able to achieve using traditional Web Services, i.e., other protocols support and even duplex communication. With WCF, we can define our service once and then configure it in such a way that it can be used via HTTP, TCP, IPC, and even Message Queues. We can even configure WCF services to create REST services too i.e., WCF REST Services.

The problem with using WCF Restful Services is that we need to do a lot of configurations in a WCF Service to make it a Restful Service. WCF is suited for scenarios where we want to create a service that should support special action such as one way messaging, message queues, duplex communication or the services that need to conform to WS\* specifications, whereas using WCF for creating restful services, that will provide fully resource-oriented services over HTTP is a little complex. Still WCF is the only option for creating the Restful services if there is a limitation of using .NET 3.5 Framework.

**Evolution of Web API:** To overcome the above problems, Microsoft came up with ASP.NET Web API to facilitate the creation of Restful Services that can provide fully resource-oriented services for broad range of clients including desktops, browsers, mobiles, and tablets. ASP.NET Web API is a framework for building REST Services easily and in a rather simple way.

**ASP.NET Web API Characteristics:**

* ASP.NET Web API is an ideal platform for building Restful Services.
* ASP.NET Web API is built on top of ASP.NET and supports ASP.NET request/response pipeline.
* ASP.NET Web API maps HTTP Verbs to method names.
* ASP.NET Web API supports different formats of response data and built-in support for JSON, XML, and BSON format.
* ASP.NET Web API can be hosted in IIS, Self-Hosted or other Web Servers that supports .NET 4.0+.
* ASP.NET Web API framework includes new “HTTPClient” class to communicate with Web API server. Http Client can be used in ASP.MVC App’s, Windows Form App’s, Console App’s, or other Apps.
* **Consuming the Web Api Service from an MVC Application using C# Code:** to test consuming the above “Web Api service” from an MVC Project, create a new “ASP.NET Web Application” project naming it as “WebApiConsumer2”, choose “Empty Project Template”, select “MVC” Check Box and click on “Create” button.
* To call Web API from an MVC Application using C# code we need to first Install a Nuget Package “Microsoft.AspNet.WebApi.Client” in our project which provides all the types for invoking a Web API Service from an “MVC Controller-Action Methods”.
* Right now, our MVC Application is not getting its data from a Database, so we don’t have LINQ or Entity Framework providing the Model Classes. Our MVC Application is getting its data from a Web API Service, so in this case we are responsible to define the Model Classes explicitly in our project and to do that add a new class under the Models folder naming it as “Student” and write the below code in it:

**ASP.NET Core**

**.NET Core vs. ASP.NET Core**

|  |  |
| --- | --- |
| **.NET Core** | **ASP.NET Core** |
| .NET Core is Open-Source and Cross-Platform. | ASP.NET Core is Open-Source and Cross-Platform. |
| .NET Core is a Runtime to execute applications which are built on it. | ASP.NET Core is a Web Framework to build Web, IoT and Mobile App’s on the top of .NET Core. |
| Install .NET Core Runtime to run applications and install .NET Core SDK to build applications. | There is no separate runtime and SDK are available for ASP.NET Core. .NET Core runtime and SDK includes ASP.NET Core libraries. |
| .NET Core 6.0 - latest version. | ASP.NET Core 6.0 - latest version. |

**Note:** There is no separate versioning for ASP.NET Core. It is the same as the .NET Core versions.

**What ASP.NET Core doesn’t have?**

If you are coming from ASP.NET 4.x, then you will not find the following things in ASP.NET Core:

* Global.asax File
* Web.config File
* HTTP Handlers and HTTP Modules
* ASP.NET Page Life-Cycle Model

**Startup.cs:** This file contains a class in it with the name “Startup” and it will serve three main purposes:

* It performs all initialization tasks like setting, application - wide constants.
* It registers all the services that are injected in this project thru the DI (Dependency Injection) container.
* It defines the middleware pipeline of your web-application.

**Note:** This class initially contains lot of code (as per the project i.e., Empty or Web App or MVC or Web API, that has just been created) from the very beginning and will become even bigger when you start adding new features to your application.

**By default, Startup class in the ASP.NET Core application includes three main parts:**

1. The Constructor, where you can initialize variables, set some configuration settings, or performs application-wide initializations. By default, the ASP.NET Core project template contains 1 line of code in the constructor which initializes the in-class property Configurations with the configuration object passed by the Dependency Injection container, so we can use it in other methods and right now the code in the constructor is as below:

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

1. The ConfigureServices method, where we register all necessary services like “Authentication/Authorization Service”, “MVC or Razor Page Services”, “Service for working with Database”, as well as we register different “Application Services to DI (Dependency Injection) Container” here and by default in an ASP.NET Core MVC Application the method contains below code in it:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

}

**Note:** the order of the services you register in “ConfigureServices” method is not important and this method is executed only once upon application start and this method contains calls such as “services.AddDbContext”, “services.AddRazorPages”, “services.AddControllersWithViews” and similar. All these methods are extension methods.

1. The Configure method, is the place where we can set up the Middleware Pipeline for our ASP.NET Core Web Application project. Unlike the services registered in the ConfigureServices (remember, their order is not important), the order of all Middleware’s defined in Configure method has crucial significance.

**What is a Middleware?**

**Ans:** ASP.NET Core introduced a new concept called Middleware. A middleware is nothing but a class which is executed on every request in ASP.NET Core Application. There will be multiple Middleware’s in an ASP.NET Core Application. It can be either Framework provided Middleware added via NuGet or our own custom Middleware. We can set the order of Middleware execution in the request pipeline. Each Middleware adds or modifies Http Request and passes control to the next Middleware component.

At the beginning of the pipeline, we need to place the “Middleware’s” that are necessary for auxiliary tasks (like logging or authentication), and that don't consume a lot of memory and processing time.

**Exception Handling Middleware:** the first line in the method defines different middleware’s for Development and Production modes i.e., if we are in the Development Mode, we define middleware’s that will catch all exceptions in the pipeline and show a special page with extra information about the error (exception message, stack trace, etc.), whereas in the Production Mode, we catch all exceptions and then re-direct the request to the specified path i.e., “Home/Error” in our case.