

ASP.NET

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. 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

Desktop Applications means, these applications must be installed on our **computer** first to consume them.

Example: MS Office, Skype Messenger, Zoom, Browsers, etc.

Web Applications means, we install these applications on a centralized machine known as **Web Server** and then clients can access the application by connecting to the **Web Server** using a **Browser** thru **Internet**.

Example: Facebook, Gmail, Amazon, Flipkart, etc.

Mobile Applications are also like **Desktop Applications** only i.e., we need to install them on our **Mobiles** or **Tablets** to consume, but they work with the help of **Internet** like a **Web Application**.

Example: WhatsApp, Swiggy, Zomato, Uber, Ola, etc.

Desktop Applications vs Web Applications:

- Web Applications need to be installed only once that to on 1 Computer only whereas Desktop App's must be installed separately on each computer.
- When we need to update a Web App's it needs to be done only on the single computer where it is installed, whereas in case of Desktop App's it needs to be done on every computer.
- Desktop App's are confined to a particular location, and they have usability constraints, whereas Web App's are convenient for the users to access them from any location using internet.
- Web Application's relies significantly on internet connectivity and speed, so absence of internet or poor connectivity can cause performance issues whereas Desktop App's are standalone in nature and hence do not face any hindrances resulting from Internet connectivity.
- Web Application is completely internet dependent, so they consume more bandwidth whereas Desktop App's are partially internet dependent, so they consume less bandwidth.
- To build Desktop Application with .NET Languages we are provided with technologies like Console App's, Windows App's, WPF (Windows Presentation Foundation) App's and same as that to build Web Applications we are provided with technologies like ASP.NET and ASP.NET Core.

Web Applications vs Mobile Applications:

- Web apps need an active internet connection to run, whereas mobile apps may work offline.
- Mobile apps have the advantage of being faster and more efficient, but they do require the user to regularly download updates. Web apps will update themselves.
- Mobile apps are developed for operating systems like iOS and Android. They can be accessed from anywhere by downloading the application on the device you want to use. Whereas the Web application can run on any OS like desktop operating systems as well as mobile operation systems also.
- Mobile applications are much safer when compared to web applications.
- The web app is less expensive when compared to mobile apps.

Desktop Applications vs Mobile Applications:

- Desktop apps run on stationary machines or laptops, while mobile apps run on mobile devices such as mobile phones or tablets.
- Mobile apps have limitations when it comes to storage, connectivity, etc.
- Mobile apps are designed to function with the phone's inbuilt features like the microphone, camera, location services, etc.
- Desktop apps are usually more flexible and agile than mobile apps.
- Mobile users interact with their devices in many ways than traditional apps. For example, mobile apps must be designed for vertical orientation, while desktop apps can be designed for landscape orientation.

What is Internet?

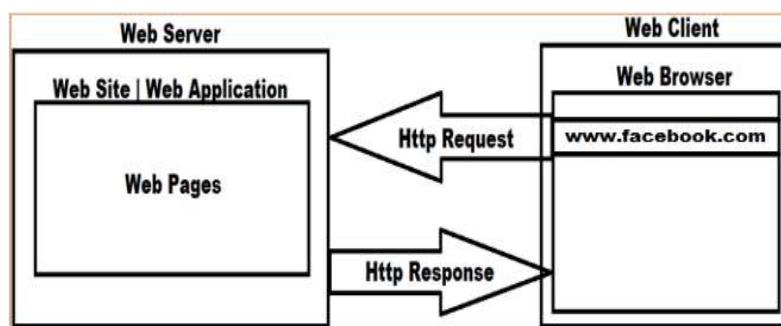
Ans: Internet is a global system of interconnected computer networks that use the standard Internet protocol suite (TCP/IP) to link several billion devices worldwide. It is a *network of networks* that consists of millions of private, public, academic, business, and government networks, of local to global scope, that are linked by a broad array of electronic, wireless, and optical networking technologies. Internet carries an extensive range of information, resources, and services, such as the inter-linked **hypertext** documents and applications of the **World Wide Web** (WWW), the infrastructure to support **email**, and **peer-to-peer** networks for **file sharing** and **telephony**.

What is World Wide Web (WWW)?

- The Web is a network of computers all over the world.
- All the computers in the web can communicate with each other.
- All the computers use a communication protocol called Http(s).

How does WWW work?

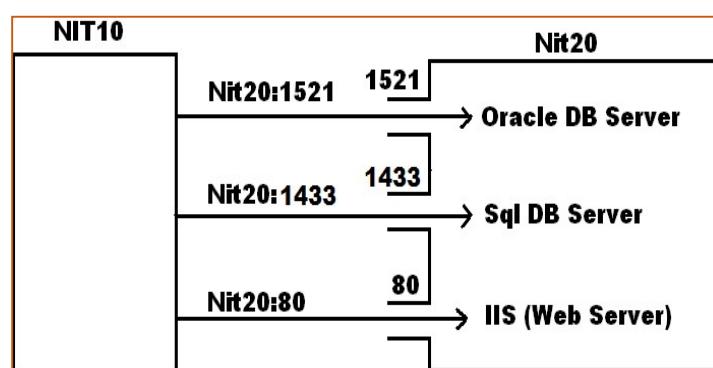
- Web information is stored in files called Web Pages.
- Collection of Web Pages is known as Web Site or Web Application.
- Web Site or Web Application is hosted on computers called Web Servers.
- Devices reading the Web Pages are called Web Clients.
- Web Clients views the Web Pages with a program called as Web Browser.



How does a Web Browser fetch a Web Page?

- A browser fetches a Web Page from a Web Server by a request and this request is a standard "HTTP Request" containing a page address known as URL (Uniform Resource Locator).
- An address or URL may look like this: <http://www.google.com:80/default.html>

Format of URL: <Protocol>://<Domain Name>:<Port>/<Request Page>



How does a Browser Display a Web Page?

- All Web Pages contain instructions for display and the Web Browser displays the page by reading those instructions.
- The most common display instructions are called HTML.

What is a Web Server?

- The collection of Web Pages is called as a Web Site or Web Application.
- To let others, view your Web Pages, you must publish your Web Site or Web Application.
- To publish your Site or Application, you must copy it to a Web Server.
- Your own PC can act as a Web Server if it is connected to a network, but the most common practice is to use a Hosting Internet Service Provider (ISP).

What is an Internet Service Provider?

- An ISP provides Internet Services.
- A common Internet service is Web Hosting.
- Web Hosting means storing your Web Site or Web Application on a Public Server.
- Web Hosting normally includes Email Services also.
- Web Hosting often includes Domain Name Registration also.

Summary:

- If you want other people to view your Web Site or Application, you must copy your Site to a Public Server.
- Even if you can use your own PC as a Web Server but it is very common to let an ISP host your Site.
- Included in Web Hosting solution you can also expect Domain Name Registration and standard Email Services.
- Hosting your Site or Application on your own Server is always an option and here are some points to consider:
 1. **Hardware Expenses:** To run a “real” Web Site or Web Application, you will have to buy some powerful **Server** hardware. Don’t expect that a **low-cost** PC will do the job. You will also need a permanent (**24 hours a day**) high-speed internet connection.
 2. **Software Expenses:** Remember that **Server-Licenses** often are higher than **Client-Licenses**. Also note that **Server-Licenses** might have limit on number of users.
 3. **Labour Expenses:** Don’t expect low **labour expenses**. You must install your **own hardware** and **software**. You also must deal with bugs and viruses and keep your server constantly running.

Benefits of using an Internet Service Provider (Public Server): renting a **Web Server** from an **ISP** is a very common option and most small companies store their **Web Site** or **Web Application** on a **Server** provided by an **ISP** only. Here are some advantages:

1. **Connection Speed:** Most ISPs have very fast connections to the Internet.
2. **Security and Stability:** ISPs are specialists on Web Hosting. Expect their Servers to have more than 99% up time, the latest software patches, and the best Virus Protection.
3. **Powerful Hardware:** ISPs often have powerful Web Servers that can be shared by several companies. You can also expect them to have an effective Load Balancing, and necessary Backup Servers.

Things to Consider with an ISP:

1. **24-hour support:** Make sure your ISP offers 24-hours support. Don’t put yourself in a situation where you cannot fix critical problems without having to wait until the next working day. Toll-free phone could be vital if you don’t want to pay for long distance calls.

2. **Daily Backup:** Make sure your ISP runs a daily backup routine; otherwise, you may lose valuable data.
3. **Traffic Volume:** Study the ISP's traffic volume restrictions. Make sure that you don't have to pay a fortune for unexpected high traffic if your Site becomes popular.
4. **Content Restrictions:** Study the ISP's content restrictions if you plan to publish pictures or broadcast video or sound and make sure that you can do it.
5. **E-mail Capabilities:** Make sure your ISP supports the Email capabilities you need.
6. **Database Access:** If you plan to use data from Databases in your Site, make sure your ISP supports the Database access you need.

What is TCP/IP?

Ans: TCP/IP stands for **Transmission Control Protocol / Internet Protocol**, is a family of protocols for communication between **computers**. It defines how **electronic devices** should be connected over the network, and how data should be transmitted between them.

- **TCP:** is responsible for **breaking down data** into **small packets** before they can be sent over a **network**, and for **assembling the packets** again when they **arrive**.
- **IP:** takes care of the **communication** between **computers**. It is responsible for **addressing, sending, and receiving** the **data packets** over the Internet.

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

1. **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**.
2. **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**.
3. **FTP:** File Transfer Protocol, takes care of **transmission** of files between **Computers**.
4. **SMTP:** Simple Mail Transfer Protocol, used to **send** or **receive** **Emails** between **Senders** and **Receivers**.

IP is Connection-Less:

- IP is a “connection-less” communication protocol.
- IP does not occupy the **communication line** between two **computers**. This **reduces** the need for **network** lines. Each **line** can be used for **communication** between many different **computers** at the same time.
- With IP, messages are broken up into small independent “packets” and sent between **computers** via the **Internet**. IP is responsible for “routing” each packet to the correct **destination**.

IP Routers:

- When an **IP Packet** is sent from a computer, it arrives at an **IP Router**.
- IP Router is now responsible for “routing” the packet to the correct **destination**, directly or **via** another **Router**.
- Communicating via IP is like **sending a long letter** as many small **postcards**, each finding its own way to the **receiver**.

IP Addresses:

- This is a **unique identification** for every **device** in the **network**.
- IP uses **32 bits or four bytes** numeric value where each **number** should be ranging between **0** to **255**.

- IP addresses are normally written as **four numbers** separated by a **period**, like this: **31.13.65.36**.
- Each **device** must have a unique **IP address** before it can **connect** to the **Internet**.
- Each **IP packet** must have an **address** before it can be sent to another **computer**.
- This is an **IP address**: **31.13.65.36**. This might be the same address: www.facebook.com

Domain Names:

- A name is much easier to remember than a **12-digit number**.
- Names used for **IP addresses** are called **Domain Names**; for example, “www.facebook.com” is a **domain name**.
- When you address a web site, like “www.facebook.com”, the name is **translated** to a **number** by a **Domain Name Server (DNS)**.
- All over the world, **DNS Servers** are connected to the **Internet**. **DNS servers** are responsible for **translating domain names** into **IP Addresses**.
- When a new **domain name** is registered together with an **IP Address**, **DNS servers** all over the world are **updated** with this information.

What is a Domain Name?

- A **domain name** is a **unique** name for a **Web Site**, like google.com, facebook.com.
- Choosing a **hosting solution** should include **domain name registration** also.
- **Domain names** must be **registered**. When **domain names** are **registered**, they are added to a **large domain name register**. In addition, information about the **Web Site**, including the **IP Address** is stored on a **DNS Server**.

What is a DNS?

- **DNS** stands for **Domain Name System**.
- A **DNS Server** is responsible for informing all other **computers** on the **Internet** about the **domain name** and the **Web Site address**.

Registering a Domain:

- Domains can be registered from **domain name registration** companies.
- These companies provide an **interface** to search for **available domain names**, and they offer a variety of **domain name extensions** that can be **registered** at the same time.

Choosing a Domain Name:

- Choosing a **domain name** is a major step for any **individual or organization**.
- New **domain name** extensions and **creative** thinking offer thousands of excellent **domain names**!
- When **choosing** a name, it is important to consider the purpose of a **domain name**, which is to provide an easy way to reach your **web site**.
- The best **domains** have the following listed characteristics: **Short, Meaningful, Clear and Exposure**:
 1. **Short** - People don't like to type! So, a short **domain name** is always easier to **type, read, and remember**.
 2. **Meaningful** - A short **domain** is nothing without **meaning**, “34i4nh.com” is not easy to **enter** or to **remember**. Select a **domain** that relates to your **site** in a way that people will understand about you or your site.
 3. **Clear** - **Clarity** is important when selecting a **domain name**. Avoid a name that is difficult to **spell** or **pronounce**.
 4. **Exposure** - Names that are **short** and **easy** to remember are an **asset**. In addition to **visitors**, also consider **search engines**. **Search engines** index your **site** and **rank** it for **relevance** against **terms** people **search** for your **sites**, consider including a **relevant search term** in your **domain**.

What is web hosting?

Ans: Web hosting is a service provided by companies (the web host) that sell or lease space on a server where you can store the files that make your website accessible on the internet. This typically requires that you own a domain, and these companies may help you in purchase one.

What is shared hosting?

Ans: Shared hosting is a popular hosting option where a provider hosts multiple websites on one physical web server. Typically, most websites don't use many server resources, so shared hosting lets providers offer stable service at a low cost. Shared hosting allows you to share hosting space and costs with others, while benefitting from the speed and space you need for your small business website.

Domain Name Servers (DNS): it is the Internet's equivalent of a phone book. They maintain a directory of domain names and translate them to IP Addresses. This is necessary because, although domain names are easy for people to remember, computers or machines, access websites based on IP Addresses only. Information from all the domain name servers across the Internet are gathered and housed at the Central Registry. Host companies and Internet Service Providers interact with the Central Registry on a regular schedule to get updated DNS information.

The Internet Assigned Numbers Authority (IANA): manages IP address space allocations globally and delegates five Regional Internet Registries (RIRs) to allocate IP address blocks to local Internet Registries like ISP's and other entities. A Regional Internet Registry (RIR) is an organization that manages the allocation and registration of Internet number resources within a particular region of the world. The Regional Internet Registry system eventually dividing the world into 5 RIR's:

- African Network Information Center (AFRINIC) for Africa.
- American Registry for Internet Numbers (ARIN) for the United States, Canada, several parts of the Caribbean region, and Antarctica.
- Asia-Pacific Network Information Centre (APNIC) for Asia, Australia, New Zealand, and neighboring countries.
- Latin America and Caribbean Network Information Centre (LACNIC) for Latin America and parts of the Caribbean region.
- Reseaux IP European's Network Coordination Centre (RIPE NCC) for Europe, Russia, Middle East, and Central Asia.

Design Patterns

Design patterns represent the best practices used by experienced **object-oriented** software developers. Design patterns are **solutions** to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period.

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps un-experienced developers to learn software design in an easy and faster way. Design patterns are solutions to software design problems you find again and again in real-world application development.

What is Gang of Four (GOF)?

In 1994, four authors **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides** published a book titled “**Design Patterns - Elements of Reusable Object-Oriented Software**” which initiated the concept of **Design Pattern** in software development. These authors are collectively known as **Gang of Four (GOF)**. The book is divided into 2 parts, first part explaining about the “**Pros and Cons**” of **Object-Oriented Programming** and in the second explaining the evolution of **23 Software Design Patterns**.

Types of Design Patterns:

As per the design pattern reference book “**Design Patterns - Elements of Reusable Object-Oriented Software**”, there are 23 design patterns which can be classified in to three categories: **Creational, Structural and Behavioral** patterns.

Creational Patterns: These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using **new** operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Patterns: These design patterns concern class and object composition. Concept of **inheritance** is used to compose **interfaces** and define ways to compose objects to obtain new functionalities.

Behavioral Patterns: These design patterns are specifically concerned with **communication** between objects.

Creational Patterns:

- Abstract Factory Creates an instance of several families of classes.
- Builder Separates object construction from its representation.
- Factory Method Creates an instance of several derived classes.
- Prototype A fully initialized instance to be copied or cloned.
- Singleton A class of which only a single instance can exist.

Structural Patterns:

- Adapter Match interfaces of different classes.
- Bridge Separates an object’s interface from its implementation.
- Composite A tree structure of simple and composite objects.
- Decorator Add responsibilities to objects dynamically.
- Facade A single class that represents an entire subsystem.
- Flyweight A fine-grained instance used for efficient sharing.
- Proxy An object representing another object.

Behavioral Patterns:

- | | |
|---------------------------|---|
| ➤ Chain of Responsibility | A way of passing a request between a chain of objects |
| ➤ Command | Encapsulate a command request as an object. |
| ➤ Interpreter | A way to include language elements in a program. |
| ➤ Iterator | Sequentially accessing the elements of a collection. |
| ➤ Mediator | Defines simplified communication between classes. |
| ➤ Memento | Capture and restore an object's internal state. |
| ➤ Observer | A way of notifying change to several classes |
| ➤ State | Alter an object's behavior when its state changes. |
| ➤ Strategy | Encapsulates an algorithm inside a class. |
| ➤ Template Method | Defer the exact steps of an algorithm to a subclass. |
| ➤ Visitor | Defines a new operation to a class without change. |
-

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.

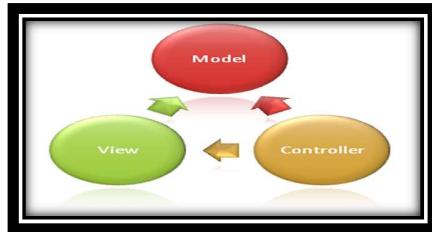
It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk and cost.

List of Architectural Patterns:

- Blackboard system
- Broker pattern
- Event - driven architecture
- Implicit invocation
- Layers
- Microservices
- Model - View - Controller (MVC)
- Presentation - abstraction - control
- Model - View - Presenter (MVP)
- Model - View - View Model (MVVM)
- Entity - component - system
- Multitier architecture (often called as 3-tier or n-tier)
- Naked objects
- Operational data store (ODS)
- Peer - to - peer
- Pipe and filter architecture
- Service - oriented architecture
- Space - based architecture

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 Data Logic or operations that should be performed by it. Data 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).

History of MVC: MVC traditionally used for **Desktop Graphical User Interfaces (GUIs)** and later this architecture has become popular for designing **Web Applications** and even **Mobile** and other **Clients**. Popular programming languages like **Java**, **.NET**, **Ruby**, **PHP**, **Python**, **Java Script**, and others have their own **MVC Frameworks** that are currently being used in **Web Application** development.

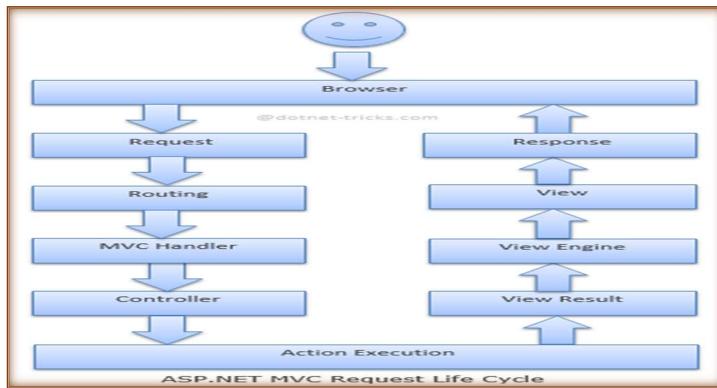
Language	MVC Framework
Java	Spring, Struts
PHP	Cake PHP, Code Igniter, Zend Framework, Laravel
Perl	Catalyst, Dancer
Python	Django, Flask, Grok
Ruby	Ruby on Rails, Nitro
Java Script	Angular, Spine
.NET	ASP.NET MVC, ASP.NET Core MVC

Trygve Reenskaug introduced **MVC** into **Smalltalk-76** while working at **Xerox Palo Alto Research Center** in the **70's**. In the **1980's**, Jim Althoff and others implemented a version of **MVC** for the **Smalltalk-80** class library.

Although originally developed for desktop computing, **MVC** has been widely adopted as architecture for **Web Applications** in major programming languages. Several **MVC Web Frameworks** have been designed that enforce the **MVC** patterns that are listed above.

MVC pattern has subsequently evolved giving rise to variants such as **hierarchical model -view - controller (HMVC)**, **model - view - adapter (MVA)**, **model - view - presenter (MVP)**, **model - view - view model (MVVM)**, and others that adapted **MVC** for different contexts.

ASP.NET MVC Request Life Cycle:



Routing: ASP.NET Routing is the first step in MVC request cycle. Basically, it is a pattern matching system that matches the request's URL against the registered URL patterns in the Route Table. When a matching pattern found in the Route Table, the Routing engine forwards the request to the corresponding "**IRouteHandler**" for that request and the default one calls the "**MvcHandler**". The routing engine returns a 404 HTTP status code against that request if the patterns are not found in the Route Table. When application starts at first time, it registers one or more patterns to the Route Table to tell the routing system what to do with any requests that match these patterns. An application has only one Route Table and this is setup in the "**Global.asax**" file of the application.

MvcHandler: The "**MvcHandler**" is responsible for initiating the real processing inside ASP.NET MVC. "**MvcHandler**" implements "**IHttpHandler**" interface and further process the request by using "**ProcessRequest**" method.

Controller: "**MvcHandler**" uses the "**IControllerFactory**" instance and tries to get an "**IController**" instance. If successful, then it calls the Execute method.

Action Execution: Once the controller has been instantiated, Controller's "**ActionInvoker**" determines which specific action to invoke on the controller. Action to be executed is chosen based on attributes "**ActionNameSelectorAttribute**" (by default method which have the same name as the action is chosen) and "**ActionMethodSelectorAttribute**" (If more than one method found, the correct one is chosen with the help of this attribute).

View Result: The action method receives user input, prepares the appropriate response data, and then executes the result by returning a result type. The result type can be "**ViewResult**", "**RedirectToRouteResult**", "**PartialViewResult**", "**RedirectResult**", "**ContentResult**", "**JsonResult**", "**FileResult**", and "**EmptyResult**".

View Engine: The first step in the execution of the **View Result** involves the selection of the appropriate **View Engine** to render the View Result. It is handled by "**IViewEngine**" interface of the view engine. By default, **ASP.NET MVC** uses "**Web Forms**" and "**Razor**" view engines. You can also register your own custom view engine to your MVC application.

View: Action method may return a text string, a binary file or a **Json** formatted data. The most important Action Result is the **View Result**, which renders and returns an HTML page to the browser by using the current view engine.

ASP.NET MVC Versions:

Date	Version
December 2007	ASP.NET MVC CTP
March 2009	ASP.NET MVC 1.0
March 2010	ASP.NET MVC 2
January 2011	ASP.NET MVC 3
August 2012	ASP.NET MVC 4
October 2013	ASP.NET MVC 5
January 2014	ASP.NET MVC 5.1
July 2014	ASP.NET MVC 5.2.0
August 2014	ASP.NET MVC 5.2.2
February 2015	ASP.NET MVC 5.2.3
February 2018	ASP.NET MVC 5.2.4
May 2018	ASP.NET MVC 5.2.5
May 2018	ASP.NET MVC 5.2.6
November 2018	ASP.NET MVC 5.2.7
April 2022	ASP.NET MVC 5.2.8
June 2022	ASP.NET MVC 5.2.9

ASP.NET Core MVC Versions:

Date	Version
August 2016	ASP.NET Core MVC 1.0.0
November 2016	ASP.NET Core MVC 1.1.0
August 2017	ASP.NET Core MVC 2.0.0
May 2018	ASP.NET Core MVC 2.1.0
December 2018	ASP.NET Core MVC 2.2.0
September 2019	ASP.NET Core MVC 3.0.0
December 2019	ASP.NET Core MVC 3.1.0
November 2020	ASP.NET Core MVC 5.0
November 2021	ASP.NET Core MVC 6.0
November 2022	ASP.NET Core MVC 7.0
November 2023	ASP.NET Core MVC 8.0

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

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 Software** 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 **Server Operating Systems** (Windows Servers, Linux, Solaris, etc.) **Database Servers** (Oracle, SQL Server, and My SQL etc.), **Application Servers**, **Web Servers**, etc., and for us what we need is a **Web Server** to provide access to our **Web Pages** to **Remote Clients**.

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**, but the most compatible **Web Server** for **ASP.NET Applications** is **IIS Web Server** from **Microsoft**.

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 the **Operating System**, and to verify whether it is installed on your machine or not, open any **Web Browser** and type in the below URL:

<http://localhost>

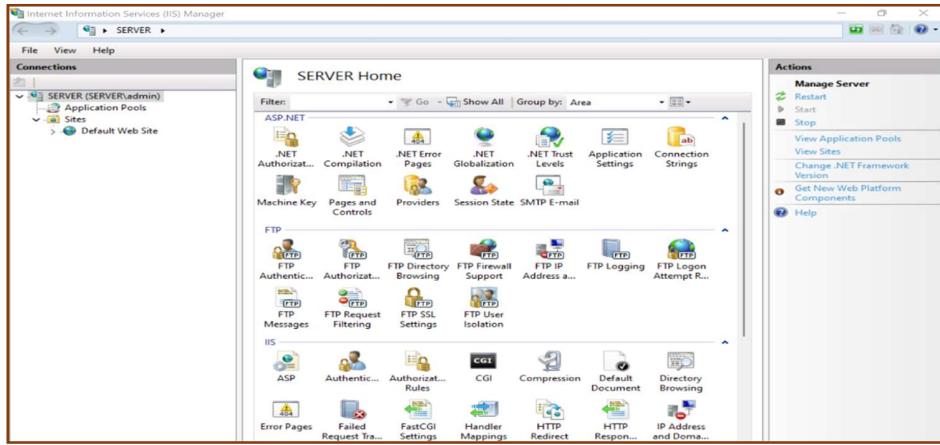
Note: If **IIS** is not installed on your machine, we get the below error:

“**HTTP Error 404. The requested resource is not found.**”

Installing IIS on our machine if not installed already: 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**) under it and click on the **Ok** button which will install **IIS** on your **machine**, then restart your machine. Once **IIS** is installed on your machine **re-verify** whether it is working or not by using the above URL.

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**".

To access Web Pages thru IIS Web Server, do the following:

Step 1: Create a **folder** on your **PC** in any drive naming it as "**MVC**".

Step 2: Create an **HTML Page** with the name "**Login.html**" with the below code and save it into "**MVC**" folder.

```
<!DOCTYPE html>
<html>
<head><title>Login Form</title></head>
<body>
<form>
<table align="center">
<caption>Login Form</caption>
<tr>
<td>User Name:</td>
<td><input type="text" id="txtName" name="txtName" /></td>
</tr>
<tr>
<td>Password:</td>
<td><input type="password" id="txtPwd" name="txtPwd" /></td>
</tr>
<tr>
<td align="center" colspan=2>
<input type="submit" value="Login" id="btnLogin" name="btnLogin" />
<input type="reset" />
</td>
</tr>
</table>
</form>
</body>
</html>
```

If we want to access Web Pages thru Web Server, we are provided with 3 different options:

Option 1: Accessing thru “Default Web Site”, because this Website is already created under IIS Web Server we can access our Web Pages thru the Site and to do that we need to copy our Web Pages to a folder that is linked with this Website i.e., “<OS Drive>:\inetpub\wwwroot” and if we copy our Web Pages into this folder we can access them thru IIS using their “Virtual Path” either from a local or a remote machine also. To test this let’s copy our “Login.html” file into this folder and then access it thru the Virtual Path of the file as following:

Local Machine => <http://localhost/Login.html> or <http://Server/Login.html>

Remote Machine => <http://Server/Login.html>

Note: for every Web Site that is created on IIS, there will be 1 associated folder on the Hard Disk and all the Web Pages of that Site should be placed into that folder, right now “Default Web Site” is mapped with the “wwwroot” folder and that is the reason why we placed our “Login.html” Page into that folder. To view the mapping folder of “Default Web Site”, right click on the “Default Web Site” in “IIS Manager” Window => select “Manage Website” and under that select “Advanced Settings” which opens a window and in that we find “Physical Path” property and beside that we find the folder that is mapped to this Web Site.

Option 2: Accessing them thru an application or virtual directory created under Default Web Site, in this case without copying all the Web Pages into “inetpub/wwwroot” folder as we did in the previous case, we can create an “Application or Virtual Directory” under “Default Web Site” and map them to our physical folder where we have saved our Web Pages i.e., “MVC” Folder.

Note: An Application or Virtual Directory is a Sub-Item under the Default Web Site mapped with a physical folder.

To create an application or virtual directory, Right Click on “Default Web Site” in “IIS Manager”, select the option “Add Application” or “Add Virtual Directory” which opens a Window and in that, in “Alias” TextBox enter a name of your choice and under “Physical Path” TextBox enter the physical path of your folder i.e., “<drive>:\MVC”.

Now following the above process, create 1 Application with the name “Site1” and create 1 Virtual Directory with the name “Site2” and map both to our physical folder i.e., “MVC”. From now we can access the page of this folder in any of the following ways:

Local Machine: <http://localhost/Site1/Login.html> Or <http://Server/Site1/Login.html>
Local Machine: <http://localhost/Site2/Login.html> Or <http://Server/Site2/Login.html>

Remote Machine: <http://Server/Site1/Login.html>
Remote Machine: <http://Server/Site2/Login.html>

Option 3: Accessing the Web Pages by creating a Site. In this case we create a new Site under IIS just like the existing site i.e., Default Web Site and map it to a physical folder, but the difference is; in this case we can give our own Host Name or Domain Name just like “localhost” which is the Host Name or Domain Name for “Default Web Site” or we can use the same Host Name i.e., “localhost” and change the Port No.

Creating a new Site by changing Port No: right click on “Sites” node in “Connections” panel and select the option “Add Website”, which opens a new window and in that window under “Site name” Textbox specify a name to the site for example: “NitSite1” and under “Physical path” TextBox specify location of our folder i.e., “<drive>:\MVC”,

now under “**Port**” Textbox the default Port will be shown as “**80**”, change to any new value like “**90**” and click on the “**OK**” button which will create the new site under **IIS**. Now we can access the **Web Pages** in our folder using the below URL:

Local Machine: <http://localhost:90/Login.html> Or <http://Server:90/Login.html>

Remote Machine: <http://Server:90/Login.html>

Creating a new Site by changing Host or Domain Name: right click on “**Sites**” node in “**Connections**” panel and select the option “**Add Website**”, which opens a new window and in that window under “**Site name**” Textbox specify a name to the site for example: “**NitSite2**” and under “**Physical path**” TextBox specify location of our folder i.e., “**<drive>:\MVC**”, now under “**Host Name**” TextBox specify a **Host Name or Domain Name**, for example “**nitsite.com**” and click on the “**OK**” button, which will create a new **Site** under **Sites**.

Note: To use the **Web Site** we need to specify the **Host Name** under a file, “**hosts**” which is in the following folder: “**C:\Windows\System32\drivers\etc**”. To do that first copy the file into a different location (because it can’t be edited in the current location), add the below code in it in the last line, save and again copy the file back to “**C:\Windows\System32\drivers\etc**” folder.

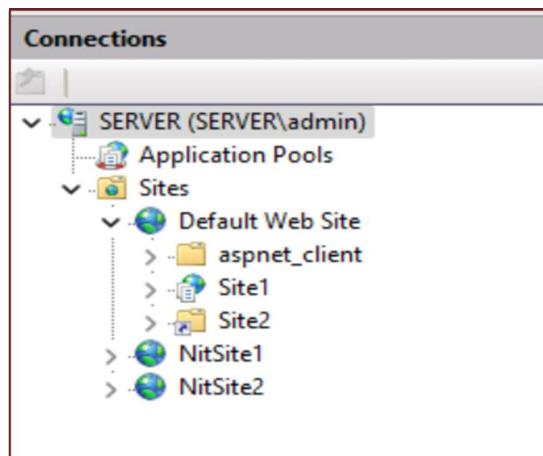
127.0.0.1 nitsite.com

Now we can access the Web Pages in our folder using the following URL:

Local Machine => <http://nitsite.com/Login.html>

Remote Machine => <http://nitsite.com/Login.html>

Note: till now we have created 1 **Application** and 1 **Virtual Directory** under “**Default Web Site**” and 2 **Sites** under the **Server** with the name “**MySite1**” and “**MySite2**” so all these 4 will be present under “**IIS Manager**” under **Connections** panel as below:



ASP.NET MVC 5

Developing an ASP.NET MVC 5 Application: This is the last version of **MVC** from **.NET Framework** and the next version is first named as **MVC 6** but later changed to **ASP.NET Core MVC** and ported to **.NET Core**. So, first let's learn working with **ASP.NET MVC 5**.

Step 1: Download and install the latest version of **Visual Studio Community Edition** i.e., VS 2022 (Version 17), from <https://visualstudio.microsoft.com/downloads> and in the downloader select the below **workloads**:

- ASP.NET and web development
- Azure development
- Data storage and processing
- Visual Studio extension development

Now select individual components tab on the top and make sure all the below are selected:

- .NET 8.0 Runtime (Long Term Support)
- .NET 7.0 Runtime (Standard Term Support)
- .NET 6.0 Runtime (Long Term Support)
- .NET 5.0 Runtime (Out of Support)
- .NET Core 2.1 Runtime
- .NET Core 3.1 Runtime
- Select all .NET Framework versions Checkbox's leaving other out of support Checkbox's.

Step 2: Creating an MVC Project and to do this, open Visual Studio 2022 => Click on “Create a new project” on RHS of the window, now in the new window opened, under “All Languages” Dropdown select “C#”, under “All Platforms” Dropdown select “Windows”, under “All Project Types” Dropdown select “Web”, now in the below list select “ASP.NET Web Application (.NET Framework)” and click on “Next” button, now in the new Window opened, under “Project Name” TextBox enter “MVCTestProject1” as name of our project, under “Location” TextBox enter “<drive>:\MVC”, under “Framework” Dropdown List, choose the latest version of “.NET Framework” i.e., “.NET Framework 4.8” and click on “Create” button which will launch a new Window with the name “Create a new ASP.NET Web Application” and under this select the following options: choose “Empty” project template, in the RHS select “MVC” CheckBox under “Add folders & core references”, uncheck all the other **Checkbox's** over there and click on the “Create” button which will create the project.

Note: Same as the above create another new project naming the project as “MVCTestProject2”, but now under “Create a new ASP.NET Web Application” Choose “MVC” project template, uncheck all the other **Checkbox's** over there and click on the “Create” button to create the project.

MVC Project Structure: when we create a new project either by choosing “Empty Project Template and select MVC CheckBox” or “MVC Project Template”, a set of files and folders are added in the project as following:

1. Connected Services: This is used for integrating **Microsoft Azure Service** into **Visual Studio**, so that we can easily host and manage our application to **Azure Portal** from **Visual Studio** it-self.

2. App_Data: this folder contains local **Data Source** files like “.mdf” files, “.xml” files, “Excel” files, etc.

3. App_Start: this folder contains a set of **files** which contains classes, and these classes get executed when the application starts execution. **App_Start** folder can contain following files in it:

1. BundleConfig.cs => Contains BundleConfig Class
2. FilterConfig.cs => Contains FilterConfig Class
3. IdentityConfig.cs => Contains IdentityConfig Class
4. RouteConfig.cs => Contains RouteConfig Class
5. Startup.Auth.cs => Contains Startup Class

Note: if we opened an “Empty Project Template” then “App_Start” folder contains only “RouteConfig.cs” file, whereas if we opened an “MVC Project Template” then “App_Start” folder contains “BundleConfig.cs”, “FilterConfig.cs” and “RouteConfig.cs” files.

4. Content: this folder contains static files like “.css” files, “image” files, etc.

Note: if we open an “Empty Project Template” then Content folder will not be existing whereas if we open an “MVC Project Template” then this folder exists with a set of “.css” (Bootstrap) files in it.

5. Controllers: this folder contains Controller classes where each Controller should be defined in a separate file.

Note: if we open an “Empty Project Template” then Controllers folder is empty, whereas if we open an “MVC Project Template” then this folder will contain 1 default Controller with the name “HomeController” under the file “HomeController.cs”.

6. Models: this folder contains Model classes i.e., classes representing the Entities and Properties representing the Attributes of Entities as well as all the Methods to manipulate the data.

Note: if we open an “Empty Project Template” or “MVC Project Template” also this folder will exist but will be empty only.

7. Scripts: this folder contains Java Script or jQuery files that are used for development of the application.

Note: if we open an “Empty Project Template” then Scripts folder will not exist, whereas if we open an “MVC Project Template” then this folder will exist with a set of “.js” (jQuery) files in it.

8. Views: this folder contains all the View files (UI) that are required for this application and the extension of these files will be “.cshtml” in case we are working with “C#” Language or else if we are working with “VB” Language then extension of the files will be “.vbhtml” and these files contains both “C# / VB” and Html code in them, and we call these files as “Razor Pages” .

Note: if we open an “Empty Project Template” then Views folder comes with a file in it i.e., “web.config”, whereas if we open an “MVC Project Template” then this folder will contain 2 sub folder in it with the names “Home” and “Shared” and these folders will contain a set of “.cshtml” files, and apart from that “Views” folder also contain “_ViewStart.cshtml” and “web.config” files.

Under the Views folder, for every Controller it will create a folder to store all the Views that are associated with that Controller, for example if there is a Controller with the name “HomeController” then there will be a “Home” folder created under Views folder to store all the Views associated with the “HomeController” class.

Under the **Views** folder, we can also have **Shared** folder containing all the **Views** that are common for all the **Controllers** in the application, for example **Layout View**, **Error View**, etc.

9. Global.asax: this is a file that contains a Global Configuration class, and, in this class, we define a set of methods: “**Application_Start**”, “**Application_End**”, “**Application_Error**” that executes for application-level actions and other methods like: “**Session_Start**” and “**Session_End**” that executes for session level actions.

10. Packages.config: this contains information of **packages** that are used under this project, so that anyone can easily understand if there are any 3rd party **packages** consumed for the development of this application.

11. Web.config: this is a configuration file for the whole application which contains configuration settings like “**App Settings**”, “**Connection Strings**”, “**Network Settings**”, “**Compiler Settings**”, etc.

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.

Adding a Controller to our MVC Project: We can add a **Controller** to an **MVC Project** in 2 different ways:

1. Manually defining Controller class.
2. Using scaffolding to define Controller class.

Option 1: Manually defining a Controller class.

Open **Solution Explorer** => right click on the **Controllers** folder => select **Add** => **New Item or Class** => now in the “**Add New Item**” window select **Class**, name it as “**TestController.cs**” and click on the “**Add**” button. Now in the class do the following:

Step 1: Import the namespace “**System.Web.Mvc**”.

Step 2: Inherit our **TestController** class with **Controller** class which should now look as below:

```
public class TestController : Controller
```

Note: Every **Controller** class contains methods in it known as **Action** methods and there should be minimum 1 **Action** method in a **Controller** class.

Step 3: Let's define 2 action methods in the class as below:

```
public string Index()
{
    return "Hello from Test Controller - Index Action Method.";
}
public string Show()
{
    return "Hello from Test Controller - Show Action Method.";
}
```

Step 4: Press F5 to run the project, which will launch the Browser and by default it displays "<http://localhost:port>" in the address bar, add "</Test/Index>" to it, to run the Index Action method or "</Test>Show>" to run Show Action method, which should look as below:

<http://localhost:port/Test/Index>
<http://localhost:port/Test>Show>

In the above URL => "Test" is the name of Controller class, and "Index or Show" are names of the Action methods we want to invoke.

How a Web Applications runs under Visual Studio?

Ans: Every Web Application requires a Web Server to run, and the job of this Web Server is to take the request from end users and send back a response, so a Web Server is mandatory to run a Web Application. To run ASP.NET Web Applications at the time of development, Visual Studio provides a built-in Web Server, using which we can run and test our applications in development environment i.e., "IIS Express". IIS Express Web Server will start when we start our Web Application in Visual Studio, and we can see that in our computer "Status Tray".

IIS Express, for unique identification of every Web Application will assign a numeric logical address to the application and we call it as "Port", which is going to be different from Project to Project and Machine to Machine. Ports are logical i.e.; they don't have any physical existence and we use them for unique identification of an application. By default, every machine will be having 65,536 ports ranging between 0 - 65535 and IIS Express will allocate any port of its choice randomly.

Note: IIS Express is a development Web Server, which the whole industry uses in development environment, and in staging and production environments we don't run Web Applications on IIS Express i.e., we will be running them using "IIS Web Server" because IIS Express can provide access to the application only from local computer and that to IIS Express will start only when we run the project whereas when we use IIS, applications can be accessed from remote computers in the network also, and it will be running independent of it-self directly under the OS, 24/7.

We can host or publish our web applications into "IIS" from Visual Studio provided "IIS" is installed on our computers. To test "IIS" is installed on our computer or not, go to Windows Search and search for "inetmgr" which will show "Internet Information Services (IIS) Manager", click on it to open.

Installing IIS on our machine if not installed: to install IIS on our computer, 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 window opened select the CheckBox "Internet Information Services" and also select each and every Sub-CheckBox under it, and click on Ok button which installs IIS on your machine.

Hosting a Web Application in IIS from Visual Studio: to do this open **Solution Explorer**, right click on the **Project** and select **Properties** which will open the “**Project Property Window**”. In the **LHS** select **Web** and then in the **RHS** we find the option “**Servers**” and in that we will find a **DropDownList** showing the value “**IIS Express**” selected, change the selection to “**Local IIS**” in the **DropDownList** which will change the “**Project URL**” in the **TextBox** below and now the value will be: <http://localhost/MVCTestProject3> and beside the **TextBox** we will find a **Button** with the caption “**Create Virtual Directory**” click on it which will **host or deploy** the application in **IIS**, click on the **Save** button in “**Standard Tool Bar**” and close “**Project Property Window**”. Now hit F5 to run the project which will display <http://localhost/MVCTestProject3> in the address bar add, “[/Test/Index](#)” to it, to run the **Index Action method** or “[/Test>Show](#)” to run **Show Action method**, which should now look as below:

<http://localhost/MVCTestProject3/Test/Index>

<http://localhost/MVCTestProject3/Test>Show>

Default Action Method and Default Controller in an MVC Application:

We can even execute the Index Action Method using the below URL also:

IIS Express: <http://localhost:port/Test>

Local IIS: <http://localhost/MVCTestProject3/Test>

Note: in the above case even if **Action** method name is not specified in the URL still it will launch “**Index**” Action method because the default **Action** method for all controllers is **Index**. Same as that the default **Controller** for the project is “**HomeController**” and to test that add a new class in the **Project** naming it as “**HomeController**” and write the below code under the class by importing “**System.Web.MVC**” namespace and inherit the class from **Controller**:

```
public string Index()
{
    return "Hello from Home Controller - Index Action Method.";
}
public string Show()
{
    return "Hello from Home Controller - Show Action Method.";
}
```

Now when we run the project it will execute the Home Controller’s Index Action method and the URL in the address bar will be as below:

IIS Express: <http://localhost:port>

Local IIS: <http://localhost/MVCTestProject3>

The below URL’s when used will give you the following results:

IIS Express:

<http://localhost:port>

//Invokes Home Controller’s Index

<http://localhost:port/Home>

//Invokes Home Controller’s Index

<http://localhost:port/Home/Index>

//Invokes Home Controller’s Index

<http://localhost:port/Home>Show>

//Invokes Home Controller’s Show

<http://localhost:port/Test>

//Invokes Test Controller’s Index

<http://localhost:port/Test/Index>

//Invokes Test Controller’s Index

<http://localhost:port/Test>Show>

//Invokes Test Controller’s Show



Local IIS:

http://localhost/MVCTestProject3	//Invokes Home Controller's Index
http://localhost/MVCTestProject3/Home	//Invokes Home Controller's Index
http://localhost/MVCTestProject3/Home/Index	//Invokes Home Controller's Index
Show">http://localhost/MVCTestProject3/Home>Show	//Invokes Home Controller's Show
http://localhost/MVCTestProject3/Test	//Invokes Test Controller's Index
http://localhost/MVCTestProject3/Test/Index	//Invokes Test Controller's Index
Show">http://localhost/MVCTestProject3/Test>Show	//Invokes Test Controller's Show

Option 2: Using Scaffolding to define Controller class.

ASP.NET **Scaffolding** is a **code generation framework** for ASP.NET **Web Applications**. From **Visual Studio 2013** Microsoft included code generators for **MVC** and **Web API** Projects. We use this **scaffolding** in our project when we want to quickly add code that interacts with **Data Models**. Using **scaffolding** will reduce the amount of time to develop **standard operations** in our project.

To add a **controller** using **scaffolding** open **Solution Explorer** => right click on **Controllers** folder => select **Add => Controller** => now in the **"Add New Scaffolded Item"** window select **"MVC 5 Controller – Empty"** template and click on the **"Add"** button which will then open a window asking for a name, enter the name as **"DemoController"** and click on **"Add"** button which adds the **Controller** class as following:

```
public class DemoController : Controller
{
    // GET: Product
    public ActionResult Index()
    {
        return View();
    }
}
```

When we use **Scaffolding** to add a **Controller**, by default the class will inherit from **"Controller"** class and this will also import **"System.Web.Mvc"** namespace. Now delete all the existing code in class and write the below code over there:

```
public string Index()
{
    return "Hello from Demo Controller - Index Action Method.";
}
public string Show()
{
    return "Hello from Demo Controller - Show Action Method.";
}
```

IIS Express:

http://localhost:port/Demo	//Invokes Demo Controller's Index
http://localhost:port/Demo/Index	//Invokes Demo Controller's Index
Show">http://localhost:port/Demo>Show	//Invokes Demo Controller's Show

Local IIS:

http://localhost/MVCTestProject3/Demo	//Invokes Demo Controller's Index
http://localhost/MVCTestProject3/Demo/Index	//Invokes Demo Controller's Index
Show">http://localhost/MVCTestProject3/Demo>Show	//Invokes Demo Controller's Show

Where is the information of default controller “Home” and default action method “Index” were specified?

Ans: Those details are present in “RouteConfig.cs” file which is present in “App_Start” folder. To verify that open “RouteConfig.cs” file and there we find a class with the name “RouteConfig” and code in the class will be as following:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

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. In an **ASP.NET Web Forms** application that does not use **routing**, an incoming request for a **URL** typically maps to a physical file that handles the request, such as an **“.aspx”** file. For example, a request for <http://localhost:port/Products.aspx?id=4> maps to a file that is named **“Products.aspx”** that contains the code and markup for rendering a **response** to the browser. The **Web Page** uses the **query string** value **“id=4”** to determine what type of content to display.

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. For example, in this URL: <http://localhost:port/Products>Show/4>, routing parser can pass the values **Products**, **Show**, and **4** to the page handler if the route is defined by using the URL pattern **“{controller}/{action}/{id}”**. The page handler would receive a dictionary collection in which the **value** associated with the **key-controller** is **Products**, the value for the **key-action** is **Show**, and the value for the **key-id** is **4**. In a request that is not managed by **URL routing**, the **“/Products>Show/4”** fragments would be interpreted as the **path** of a file in the application.

Routes: A route is a **URL Pattern** that is mapped to a **handler**. A **handler** can be a class that **processes** the **request**, such as a **Controller** in an **MVC Application**. To define a **route**, you create an instance of the **Route** class by specifying the **URL Pattern**, the **handler**, and optionally a **name** for the **route**. You add the **route** to the **application** by adding the **Route** object to the static **Routes** property of the **RouteTable** class. The **Routes** property is a **RouteCollection** object that stores all the **routes** for the **application**.

We typically do not have to write code to add routes in an MVC application. Visual Studio project templates for MVC include pre-configured URL routes. These are defined in RouteConfig class, which is defined in “RouteConfig.cs” file and then registered in “Application_Start” method of “MVCApplication” class defined under “Global.asax” file by calling “RouteConfig.RegisterRoutes”.

URL Patterns: This can contain literal values and variable placeholders (referred to as URL Parameter's). The literals and placeholders are in segments of the URL which are delimited by the slash (/) character. When a request is made, the URL is parsed into segments and placeholders, and the variable values are provided to the request handler. This process is like the way the data in query strings is parsed and passed to the request handler. In both cases variable information is included in the URL and passed to the handler in the form of “key-value” pairs. For query strings both the keys and the values are in the URL. For routes, the keys are the placeholder names defined in the URL Pattern, and only the values are in the URL.

In an URL Pattern, you define placeholders by enclosing them under curly braces {}. You can define more than one placeholder in a segment, but they must be separated by a literal value. For example, {language}-{country}/{action} is a valid route pattern. However, {language}{country}/{action} is not a valid pattern, because there is no literal value or delimiter between the placeholders. Therefore, routing cannot determine where to separate the value for language placeholder from the value for the country placeholder.

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:

```
routes.MapRoute(  
    name: "Student",  
    url: "NIT/Students",  
    defaults: new { controller = "Student", action = "Index" }  
)
```

Now we have added a new route with the name Students, so MVC Framework will first evaluate this route to check if the incoming request satisfy the URL pattern or not, and if not satisfied then it will go to “default” route which is existing.

Now add a new Controller in the Controllers folder using “Scaffolding”, name it as “StudentController” and re-write the existing Index action method in the class as following:

```
public string Index()  
{  
    return "Hello from Student Controller - Index Action method.";  
}
```

URL to access the above action method should be as following:

IIS Express: <http://localhost:port/NIT/Students>

Local IIS: <http://localhost/MVCTestProject3/NIT/Students>

Note: while defining multiple routes it's better to define the “**default**” route in the last otherwise all incoming requests will be handled by “**default**” route only and to understand that change the sequence of the routes we defined in “**RouteConfig**” class and try to access the **Student Controller** with the above URL will give you “**Not Found – 404**” error.

Passing parameters to Controller's Action methods: we can define parameters to **Action** methods of **Controllers** in different ways like using **Route Parameters** and **Query Strings**.

Using Route Parameters: by default, we can pass 1 parameter to any **action** method i.e., “**id**” because it was defined in “**default route**” under the class “**RouteConfig**” and to check that, open “**RouteConfig.cs**” file and watch the value of “**URL**” which will be as following:

url: "{controller}/{action}/{id}"

Note: by default, this “**id**” is **optional**, and we can find that under “**defaults**” in the next line.

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

If required we can send this “**id**” value to the action method and that value can be used inside the action method and to use it, Action method should have “**id**” as parameter.

To test this, let's add a new controller under our current project i.e., “**MVCTestProject3**” with the name “**ParamsController**”, delete the existing **Index** action method and write the below **methods** under the class:

```
//Passing value to id is mandatory
public string Index1(int id)
{
    return "The value of id is: " + id;
}

//Passing value to id is optional and if not passed value will be "0"
public string Index2(int id = 0)
{
    return "The value of id is: " + id;
}

//Passing value to id is optional and if not passed value will be "null"
public string Index3(int? id)
{
    return "The value of id is: " + id;
}
```

Execute the above methods as following:

IIS Express:

http://localhost:port/Params/Index1	//Invalid
http://localhost:port/Params/Index1/100	//Valid
http://localhost:port/Params/Index2	//Valid, but value of id is 0
http://localhost:port/Params/Index2/200	//Valid
http://localhost:port/Params/Index3	//Valid, but value of id is null
http://localhost:port/Params/Index3/300	//Valid

Local IIS:

http://localhost/MVCTestProject3/Params/Index1	//Invalid
http://localhost/MVCTestProject3/Params/Index1/100	//Valid
http://localhost/MVCTestProject3/Params/Index2	//Valid, but value of id is 0
http://localhost/MVCTestProject3/Params/Index2/200	//Valid
http://localhost/MVCTestProject3/Params/Index3	//Valid, but value of id is null
http://localhost/MVCTestProject3/Params/Index3/300	//Valid

Note: the parameter name in all the **3 Index** action methods should be “**id**” only (not case sensitive and can be of any type) because it is the same name we have in “**RouteConfig.cs**” file, so we can’t change it in our action methods and if we try to use a different name to the parameter other than “**id**” then the value we passed thru the route will not be taken by the action method and to test that define 2 new action methods in the “**ParamsController**” class as below:

```
//Passing value to x is mandatory and if passed also it will not take it because of parameter name mis-match
public string Index4(int x)
{
    return "The value of id is: " + x;
}

//Passing value to x is optional and if passed also it will not take it because of parameter name mis-match
public string Index5(int? x)
{
    return "The value of id is: " + x;
}
```

Execute the above methods as following:

IIS Express:

http://localhost:port/Params/Index4	//Invalid because there is no value for x
http://localhost:port/Params/Index4/400	//Invalid because the value is not taken to x
http://localhost:port/Params/Index5	//Valid and the value for x is null
http://localhost:port/Params/Index5/500	//Valid, but now also the value for x is null only

Local IIS:

http://localhost/MVCTestProject3/Params/Index4	//Invalid because there is no value for x
http://localhost/MVCTestProject3/Params/Index4/400	//Invalid because the value is not taken to x
http://localhost/MVCTestProject3/Params/Index5	//Valid and the value for x is null
http://localhost/MVCTestProject3/Params/Index5/500	//Valid, but now also the value for x is null only

The name “**Id**” which is defined in “**RouteConfig**” under the default route is not specified to be of a particular type so while using it in our action method we can specify any type to it and to test that define a new action method in the “**ParamsController**” class as following:

```
//Passing value to Id is option because it is defined as type string & strings are by default Nullable (reference types)
public string Index6(string Id)
{
    return "The value of id is: " + Id;
}
```

Execute the above method as following:

IIS Express:

http://localhost:port/Params/Index6	//Valid and value of Id is null
Hello">http://localhost:port/Params/Index6>Hello	//Valid and value of Id is Hello
http://localhost:port/Params/Index6/600	//Valid and value of Id is 600
http://localhost:port/Params/Index6/true	//Valid and value of Id is true
http://localhost:port/Params/Index6/34.56	//Invalid, because of the special character decimal

Local IIS:

http://localhost/MVCTestProject3/Params/Index6	//Valid and value of Id is null
Hello">http://localhost/MVCTestProject3/Params/Index6>Hello	//Valid and value of Id is Hello
http://localhost/MVCTestProject3/Params/Index6/600	//Valid and value of Id is 600
http://localhost/MVCTestProject3/Params/Index6/true	//Valid and value of Id is true
http://localhost/MVCTestProject3/Params/Index6/34.56	//Invalid, because of the special character decimal

Passing multiple parameters to Action method: If we want to pass **multiple** parameters to **Action** methods, we can change the default route “URL” in **RouteConfig** class, for example if we want to have **2 parameters** to our action methods **re-write** the “URL” as following:

url: "{controller}/{action}/{id}/{name}"

Now in the below, change the defaults as following:

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional, name = UrlParameter.Optional }

Now we can define action methods to accept 2 parameters with the names “id” & “name” and to test that add 2 new Action methods under our “ParamsController” class as following:

```
//Passing value to Id is mandatory and Name is optional
public string Index7(int Id, string Name)
{
    return $"Value of Id is: {Id} and value of Name is: {Name}";
}

//Passing values to Id and Name are optional
public string Index8(int? Id, string Name)
{
    return $"Value of Id is: {Id} and value of Name is: {Name}";
}
```

Execute the above methods as following:

IIS Express:

http://localhost:port/params/Index7	//Invalid because Id is mandatory parameter
http://localhost:port/params/Index7/700	//Valid, in this case Name will be null value
http://localhost:port/params/Index7/700/Raju	//Valid
http://localhost:port/Params/Index7/Raju/700	//Invalid because parameter values not in order
http://localhost:port/Params/Index7?Id=700&Name=Raju	//Valid, as we are passing values by specifying names
http://localhost:port/Params/Index7?Name=Raju&Id=700	//Valid, as we are passing values by specifying names

Local IIS:

http://localhost/MVCTestProject3/params/Index7	//Invalid because Id is mandatory parameter
http://localhost/MVCTestProject3/params/Index7/700	//Valid, in this case Name will be null value
http://localhost/MVCTestProject3/params/Index7/700/Raju	//Valid
http://localhost/MVCTestProject3/Params/Index7/Raju/700	//Invalid because parameter values not in order
http://localhost/MVCTestProject3/Params/Index7?Id=700&Name=Raju	//Valid
http://localhost/MVCTestProject3/Params/Index7?Name=Raju&Id=700	//Valid

IIS Express:

http://localhost:port/params/Index8	//Valid, in this case Id & Name will be null value
http://localhost:port/params/Index8/800	//Valid, in this case Name will be null value
http://localhost:port/params/Index8/800/Raju	//Valid
http://localhost:port/Params/Index8/Raju/800	//Valid, in this case id will be null value
http://localhost:port/Params/Index8?Name=Raju	//Valid, in this case id will be null value
http://localhost:port/Params/Index8?Id=800&Name=Raju	//Valid, as we are passing values by specifying names
http://localhost:port/Params/Index8?Name=Raju&Id=800	//Valid, as we are passing values by specifying names

Local IIS:

http://localhost/MVCTestProject3/params/Index8	//Valid
http://localhost/MVCTestProject3/params/Index8/800	//Valid
http://localhost/MVCTestProject3/params/Index8/800/Raju	//Valid
http://localhost/MVCTestProject3/Params/Index8/Raju/800	//Valid
http://localhost/MVCTestProject3/Params/Index8?Name=Raju	//Valid
http://localhost/MVCTestProject3t/Params/Index8?Id=800&Name=Raju	//Valid
http://localhost/MVCTestProject3/Params/Index8?Name=Raju&Id=800	//Valid

Using Query String: by using **Query String's** also we can pass values to **Action** methods and to test that add a new **Action** method into the “**ParamsController**” class as following:

```
public string Index9(int Pid, string Pname, double Price)
{
    return $"Pid: {Pid}; Pname: {Pname}; Price: {Price}";
}
```

Now run the application by using the following URL:

IIS Express:

http://localhost:port/Params/Index9/101/Shoes/3500	//Invalid
http://localhost:port/Params/Index9?Pid=101&Pname=Shoes&Price=3500	//Valid
http://localhost:port/Params/Index9?Pname=Shoes&Price=3500&Pid=101	//Valid
http://localhost:port/Params/Index9?Price=3500&Pid=101&Pname=Shoes	//Valid

Local IIS:

http://localhost/MVCTestProject3/Params/Index9/101/Shoes/3500	//Invalid
http://localhost/MVCTestProject3/Params/Index9?Pid=101&Pname=Shoes&Price=3500	//Valid
http://localhost/MVCTestProject3/Params/Index9?Pname=Shoes&Price=3500&Pid=101	//Valid
http://localhost/MVCTestProject3/Params/Index9?Price=3500&Pid=101&Pname=Shoes	//Valid

Note: Without defining any **parameters** to the **Action** method also we can read **Query String** values in our code, to test that add a new **Action** method into the “**ParamsController**” class as following:

```
public string Index10()
{
    int Pid = int.Parse(Request.QueryString["Pid"]);
    string Pname = Request.QueryString["Pname"];
    double Price = double.Parse(Request.QueryString["Price"]);
    return $"Pid: {Pid}; Pname: {Pname}; Price: {Price}";
}
```

Now run the application by using the following URL:

IIS Express:

http://localhost:port/Params/Index10/101/Shoes/3500	//Invalid
http://localhost:port/Params/Index10?Pid=101&Pname=Shoes&Price=3500	//Valid
http://localhost:port/Params/Index10?Pname=Shoes&Price=3500&Pid=101	//Valid
http://localhost:port/Params/Index10?Price=3500&Pid=101&Pname=Shoes	//Valid

Local IIS:

http://localhost/MVCTestProject3/Params/Index10/101/Shoes/3500	//Invalid
http://localhost/MVCTestProject3/Params/Index10?Pid=101&Pname=Shoes&Price=3500	//Valid
http://localhost/MVCTestProject3/Params/Index10?Pname=Shoes&Price=3500&Pid=101	//Valid
http://localhost/MVCTestProject3/Params/Index10?Price=3500&Pid=101&Pname=Shoes	//Valid

Note: In this case “**id**” and “**name**” parameters in **RouteConfig** class will not have any impact because both are **optional**.

Now let's write another 2 methods to understand about **Query String's**, 1 **with-out parameters** and 1 with **parameters**.

```
public string Validate1()
{
    string Name = Request["Name"];
    string Pwd = Request["Pwd"];
    if (Name == "Raju" && Pwd == "Admin")
        return "Valid User";
    else
        return "Invalid User";
}
public string Validate2(string Name, string Pwd)
{
    if (Name == "Raju" && Pwd == "Admin")
        return "Valid User";
    else
        return "Invalid User";
}
```

We can execute the above methods as following:

IIS Express:

http://localhost:port/Params/Validate1?Name=Raju&Pwd=Admin	//Valid
http://localhost:port/Params/Validate1?Pwd=Admin&Name=Raju	//Valid
http://localhost:port/Params/Validate2?Name=Raju&Pwd=Admin	//Valid
http://localhost:port/Params/Validate2?Pwd=Admin&Name=Raju	//Valid

Local IIS:

http://localhost/MVCTestProject3/Params/Validate1?Name=Raju&Pwd=Admin	//Valid
http://localhost/MVCTestProject3/Params/Validate1?Pwd=Admin&Name=Raju	//Valid
http://localhost/MVCTestProject3/Params/Validate2?Name=Raju&Pwd=Admin	//Valid
http://localhost/MVCTestProject3/Params/Validate2?Pwd=Admin&Name=Raju	//Valid

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:

1. **Action** methods must be **public**, so every **public** method in a **Controller** class is an **Action** method only.
2. **Action** methods cannot be **static** because behind the screen instance of the **Controller** class is used for calling the **Action** methods.
3. 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:

We can execute the above methods following:

IIS Express:

http://localhost:port/Params/SayHello1
http://localhost:port/Params/SayHello2?Name=Raju

Local IIS:

http://localhost/MVCTestProject3/Params/SayHello1
http://localhost/MVCTestProject3/Params/SayHello2?Name=Raju

4. If we want to define any **non-action** methods in a controller class, make sure they are not **public** or else **decorate** them with “**NonAction**” attribute and in this case when we try to access those methods from browser we get “**404 Not Found**” error.

```
<private or internal or protected or private protected or protected internal> string Display()
{
    return "Non-Action Method";
}

Or

[NonAction]
public string Display()
{
    return "Non-Action Method";
}
```

5. **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 type** and under this **class** there are a set of **child classes** and we call all those **classes** as **Action Result's** only, and we can use any of those **child classes** as a **return type** of our **Action** method.

List of ActionResult child classes is:

- **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 all the required logic here
    -return an ActionResult
}
```

Note: 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. The **extension** of a **View** will be “**.cshtml**” in case the programming language is “**C#**” or “**.vbhtml**” in case the programming language is “**VB.NET**”.

6. 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 in ASP.NET MVC are processed by “View Engines” to render the results and we are provided with a support of 2 different View Engines, those are:

1. Web Forms Engine
2. 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.cs” or “.aspx.vb”.

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 to use syntax.

Sample Web Form page with a for loop:

```
<%
for(int i=1;i<=10;i++)
{
%>
<h3>Hello World</h3>
<%
}
%>
```

Sample Razor page with a for loop:

```
@{
for(int i=1;i<=10;i++)
{
    <h3>Hello World</h3>
}
```

Note: apart from the above 2 View Engines, ASP.NET MVC also supports many other third-party View Engines also like “NHaml”, “Brail”, “NDjango”, “Spark”, “Hasic”, etc.

Creating Action methods returning ViewResult: create a new ASP.NET Web Application project naming it as “MVCActionResults”; choose “Empty Project Template”, check “MVC” CheckBox and click on the Create Button.

Now add a Controller in to the Controllers folder of the project naming it as “HomeController” and by default the class contains a method with the name “Index” and its return type will be “ActionResult”, change it as “ViewResult” (not mandatory) or leave the same and add 2 more Action methods in the class as following:

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

```

public ViewResult Login()
{
    return View();
}

```

Adding Views for Action Methods: we place **Views** under **Views** folder of the project and whenever a **Controller** is added to the project using **Scaffolding**, automatically a new folder gets added to the **Views** folder with the name of that **Controller**, so under the **Views** folder we will now find a folder with the name “**Home**” as we added a controller with the name “**Home**” and all **Views** that are associated with this **Controller** are generally placed into the **Home** folder.

Now let's add 3 **Views** i.e., 1 for each **Action** method and to do that right click on the **Home** folder, select **Add => “View”**, this will open a new window with the name “**Add New Scaffolded Item**”, in that select “**MVC 5 View**” and click “**Add**” which opens a new window “**Add View**”, in that window under “**View Name**” TextBox specify the view name as “**Index**” (will by default show the name as “**View**”), leave the “**Template**” TextBox with its existing value i.e., “**Empty (without model)**”, un-check all other **Checkbox’s** on the screen and click on “**Add**” button which will add the **View** with some **HTML Code** in it.

Now in the view file write the below code under the “<div>” tag:

```

<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<h2>Click on the links below to navigate:</h2>
<h3>
<a href="/Home/Register">Register</a><br />
<a href="/Home/Login">Login</a><br />
<a href="/Home/ForgotPassword">Forgot Password</a><br />
<a href="/Home/ResetPassword">Reset Password</a><br />
<a href="/Home/Contact">Contact Us</a><br />
<a href="/Home/Mission">Mission</a><br />
<a href="/Home/About">About Us</a><br />
</h3>

```

Add another view same as the above with the name Register and write the below code in its <div> tag:

```

<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<h3 style="text-align:center;text-decoration:underline">Registration Page</h3>
<table align="center">
<tr><td>Name:</td><td><input type="text" id="txtName" name="txtName" /></td></tr>
<tr><td>User Id:</td><td><input type="text" id="txtUid" name="txtUid" /></td></tr>
<tr><td>Password:</td><td><input type="password" id="txtPwd" name="txtPwd" /></td></tr>
<tr><td>Confirm Password:</td><td><input type="password" id="txtCPwd" name="txtCPwd" /></td></tr>
<tr><td>Mobile:</td><td><input type="tel" id="txtMobile" name="txtMobile" /></td></tr>
<tr><td>Email Id:</td><td><input type="email" id="txtEmail" name="txtEmail" /></td></tr>
<tr>
<td colspan="2" align="center">
<input type="submit" id="btnRegister" value="Register" />
<input type="reset" id="btnReset" value="Reset" />

```

```

</td>
</tr>
</table>
<h4 style="text-align:center;color:red">
Click here to go to <a href="/Home/Index">Home Page.</a>
</h4>

```

Now add another view same as the above with the name Login and write the below code in its <div> tag:

```

<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<h3 style="text-align:center;text-decoration:underline">Login Page</h3>
<table align="center">
<tr><td>User Id:</td><td><input type="text" id="txtUid" name="txtUid" /></td></tr>
<tr><td>Password:</td><td><input type="password" id="txtPwd" name="txtPwd" /></td></tr>
<tr>
<td colspan="2" align="center">
<input type="submit" id="btnLogin" value="Login" />
<input type="reset" id="btnReset" value="Reset" />
</td>
</tr>
</table>
<h4 style="text-align:center;color:red">
Click here to go to <a href="/Home/Index">Home Page.</a>
</h4>

```

Now run the project which will launch the “Index” view first because we are already aware that, default Controller is Home and default Action method is Index (listed in RouteConfig class). Index View will provide Links for launching Login and Register views, click on them to launch the corresponding Views.

Note: while launching a View we don’t require to specify the View Name in the Helper method => “View()” because View names matches with the Action method names.

Can the Action method name and View name be different?

Ans: Yes, Action method name and View name can be different, and if they are different, we need to explicitly pass View name or path of that View as a parameter to the Helper method. To test this, add 2 new Views naming them as “ForgotPwd” and “ResetPwd”.

Write the below code under “<div>” tag of “ForgotPwd.cshtml” file:

```

<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<div style="text-align:center;background-color:cyan">
Forgot your password? Enter your registered Email Id, to receive a reset password link.
<br />
Email Id: <input type="email" id="txtEmail" name="txtEmail" />
<input type="submit" id="btnSubmit" value="Submit" />
</div>
<h4 style="text-align:center;color:red">
Click here to go to <a href="/Home/Index">Home Page.</a>
</h4>

```

Write the below code under “<div>” tag of “ResetPwd.cshtml” file:

```
<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<div style="text-align:center;background-color:cyan">
    <h4>Reset Password</h4>
    <table align="center">
        <tr><td>New Password:</td><td><input type="password" id="txtPwd" name="txtPwd" /></td></tr>
        <tr><td>Confirm Password:</td><td><input type="password" id="txtCPwd" name="txtCPwd" /></td></tr>
        <tr>
            <td colspan="2" align="center">
                <input type="submit" id="btnSubmit" value="Submit" />
                <input type="reset" id="btnReset" value="Reset" />
            </td>
        </tr>
    </table>
</div>
<h4 style="text-align:center;color:red">Click here to go to <a href="/Home/Index">Home Page.</a></h4>
```

Now go to “HomeController” class and add 2 new Action Methods in the class as following:

```
public ViewResult ForgotPassword()
{
    return View("ForgotPwd");
}
public ViewResult ResetPassword()
{
    return View("~/Views/Home/ResetPwd.cshtml");
```

Note: in the above case **Action** method names are not matching with **View** names so we are explicitly passing **View** name as parameter to the **Action** method and that can be done in any of the above 2 ways.

Is it mandatory to place the View exactly under the folder representing the Controller?

Ans: No, it is not **mandatory** i.e., we can place them in **“Shared”** folder also which will be present inside of the **Views** folder. To test this, add a new folder under **Views** folder naming it as **“Shared”** and add a **View** into the folder naming it as **“Contact.cshtml”** and write the below code under its **“<div>”** tag:

```
<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<div style="background-color:cyan">
    <fieldset style="border: 5px solid coral">
        <legend>Contact Us:</legend>
        &nbsp;&nbsp;Phone: 2374 6666 <br />
        &nbsp;&nbsp;Whatsapp: 81791 91999 <br />
        &nbsp;&nbsp;Email: info@nareshit.com <br />
        &nbsp;&nbsp;Website: www.nareshit.com <br />
        &nbsp;&nbsp;Address: 2nd Floor, Durga Bhavani Plaza, Satyam Theatre Road, Ameerpet, Hyderabad - 500016
    </fieldset>
</div>
<h4 style="text-align:center;color:red">Click here to go to <a href="/Home/Index">Home Page.</a></h4>
```

Now go to “HomeController” class and add a new Action Methods in the class as following:

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

Note: in the above case the **View** gets launched even if it is not present inside of the **Home** folder because **View Engine** will first search in the **Home** folder and if not found, then it will search in the **Shared** folder to find the **View**. Apart from the **Shared** folder we can also place it in any other folders also but if we do so, we need to explicitly specify path of the **View** to the **Helper** method, and to test this add another new folder in **Views** folder with the name as “**Test**”, and under the new folder add a View naming it as “**Mission.cshtml**” and write the below code under its “**<div>**” tag:

```
<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<div style="background-color:burlywood">
    <h3 style="text-align: center;color:chartreuse">Our Mission</h3>
    <p style="text-align: justify; font-size:16px; text-indent:50px; color:brown">To enrich the knowledge & skill sets of young software engineers by providing value added training in the areas of Software Development & Testing. To serve the industries by providing trained human resources in the above areas. To provide quality Software Training and Consulting Services in the areas of J2EE, .NET, ERP, Database Administration, Testing, Content Management with Live Projects.
    </p>
</div>
<h4 style="text-align:center;color:red">Click here to go to <a href="/Home/Index">Home Page.</a></h4>
```

Now go to “HomeController” class and add a new Action method in the class as following:

```
public ViewResult Mission()
{
    return View("~/Views/Test/Mission.cshtml");
}
```

What is the default View Engine in an MVC 5 Application?

Ans: The default **View Engine** in an **MVC5 Application** is “**Web Form Engine**”, so if a **View** page is existing with both “**.aspx**” as well as “**.cshtml**” extensions the first preference goes to “**.aspx**” only.

To test this, add 2 new **Views** into the **Home** folder of **Views** folder with the name “**About.aspx**” and “**About.cshtml**”.

Note: From **MVC5** the support for adding “**Web Form Views**” is removed in **Visual Studio**, so we should explicitly add an “**ASP.NET Web Form**” and make the necessary changes to it to make it a “**Web Form View Page**” and to do that, right click on the **Home** folder under **Views** Folder, select **Add => New Item** and in the **New Item Window** select “**Web Form**”, name it as “**About.aspx**” and click on “**Add**” button to add a “**Web Form**” and then write the below code in “**About.aspx**” file under its “**<div>**” tag:

```
<h1 style="text-align: center; color: red; text-decoration: underline">Naresh I Technologies</h1>
<div style="background-color: cyan">
```

```

<h3 style="text-align: center">About page created using Web Form View Engine.</h3>
<p style="text-align: justify; font-size: 16px; text-indent: 50px; color: coral">Naresh I Technologies (Pronounced: NareshIT) is a leading software training institute providing Software Training, Project Guidance, IT Consulting and Technology Workshops. Using our enhanced global software training delivery methodology, innovative software training approach and industry expertise, we provide high-value corporate training services that enable our clients to enhance business performance, accelerate time-to-market, increase productivity, and improve customer service. We serve Global 100 companies and the leading software vendors in Banking & Financial Services, Insurance, Telecommunications, Technology and Media, Information & Education industries. We design and mentor human resources for our clients who create competitive advantage. Founded in 2004 and headquartered in Hyderabad, India, we have offices and training institutes throughout India.
</p>
</div>
<h4 style="text-align: center; color: red">Click here to go to <a href="/Home/Index">Home Page.</a></h4>

```

Now go to “About.aspx.cs” and to do that, right click on “About.aspx” and select the option “View Code” which will take us to “About.aspx.cs” and there we will find a class “About” inheriting from “System.Web.UI.Page”, change that as “System.Web.Mvc.ViewPage”.

Now add another view in home folder naming it as “About.cshtml” and write same code as above View in the “<div>” tag but change the “Inner Html” of “<h3>” tag in the top as: **About page created using Razor View Engine.**

Now go to HomeController class and add a new Action Methods in the class as following:

```

public ViewResult About()
{
    return View();
}

```

Now launch Index View and click on the “About Page” hyper link and this will launch “About.aspx” page because by default it will search for “.aspx” pages first, whereas if we want “About.cshtml” page to be launched we need to pass the path of “About.cshtml” file as parameter to Helper method and to test that re-write code in “About” action method as following:

```
return View("~/Views/Home/About.cshtml");
```

Without passing the address of View Page to Helper Method also, we can launch the “.cshtml” View and to do that we need to either clear all the “View Engines” and then freshly add “Razor View Engine” or remove the “Web Form View Engine” to make “Razor View Engine” as default View Engine. To do that go to “Global.asax” file and write the below code in top of “Application_Start” method:

```

ViewEngines.Engines.Clear();
ViewEngines.Engines.Add(new RazorViewEngine());
Or
ViewEngines.Engines.Remove(ViewEngines.Engines.OfType<WebFormViewEngine>().FirstOrDefault());

```

Note: Now even if the code under “About” action method is “return View();” also, it will launch “About.cshtml” only and in this case if we try to launch “About.aspx” View Page by specifying the explicit path also, we get an error because “WebFormViewEngine” support has been removed by us.

Can an Action method return different views based on a condition?

Ans: Yes, it is possible to return different **Views** by an **Action** method based on a condition. To test this, add 2 new **Views** under **Home** folder of **Views** with the names “**Show1.cshtml**” and “**Show2.cshtml**”, and write the below code under their “**<div>**” tags:

Show1.cshtml:

```
<h1 style="text-align:center;color:red;text-decoration:underline">Naresh I Technologies</h1>
<div style="background-color: cyan"><h3 style="text-align: center">This is Show1 View Page</h3></div>
<h4>Click here to go to <a href="/Home/Index">Home Page.</a></h4>
```

Show2.cshtml:

```
<h1 style="text-align: center; color: red">Naresh I Technologies</h1>
<div style="background-color: cyan"><h3 style="text-align: center">This is Show2 View Page</h3></div>
<h4>Click here to go to <a href="/Home/Index">Home Page.</a></h4>
```

Now under “HomeController” class add a new Action method as following:

```
public ViewResult Show(int id) {
    if (id == 1)
        return View("Show1");
    else
        return View("Show2");
}
```

Now go to “Index.cshtml” and add the below links over there:

```
<a href="/Home>Show/1">Show1 View</a><br />
<a href="/Home>Show/2">Show2 View</a>
```

MVC Action Selectors: these are **attributes** that can be applied on an **Action** method, and they help the **View Engine** to select the correct **Action Method** to handle the request.

We have 3 Action Selectors and those are:

1. **ActionName**
2. **NonAction**
3. **ActionVerbs**

ActionName: this attribute is used to specify a different name to any **Action** method than its **actual name**. We use this attribute when we want an **Action** method to be called with a different name instead of the actual name of the method. Generally used in-case we want to overload **Action** methods. For Example:

```
[ActionName("Launch")]
public ViewResult LaunchViewPageLoadingDataFromDatabase()
```

Non-Action: this attribute indicates a **public** method of the **Controller** is not an **Action** method i.e., when we use this attribute, then the **public** method in the **Controller** class will not be treated as an **Action** method and we can't call this method from **browsers** using a **URL**. For Example:

```
[NonAction]
public string SayHello()
```

ActionVerbs: this attribute is used on an **Action** method when we want to handle different type of **HTTP Requests**. MVC framework provides various action verbs like: “**HttpGet**”, “**HttpPost**”, “**HttpPut**”, “**HttpDelete**”, “**HttpOptions**” and “**HttpPatch**”. You can apply one or more action verbs on an action method to handle different **HTTP Requests**. If we don't apply any **action verb** on an **action method**, then by default it handles “**HttpGet**” requests. In our application if we have defined 2 **Login** Action methods and those methods are decorated with [**HttpGet**] and [**HttpPost**] **ActionVerbs**, then whenever we send the first request (**get request**) to “**Login**” Action method then it will invoke the method which is decorated with [**HttpGet**] Action Verb and launches the corresponding **View**. When we fill in the credentials and click on the “**Login**” button it will send a “**Post**” or “**Postback**” request and will then invoke “**Login**” Action method which is decorated with [**HttpPost**] Action Verb. For Example:

```
[HttpGet]  
public ViewResult Login() //Responds for Get Request  
  
[HttpPost]  
public ViewResult Login(string Name, string Password) //Responds for Post Request
```

Razor Programming

From **MVC 3.0**, Microsoft has 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:

1. Single Line Statements.
2. In-Line Statements.
3. Multi Line Statements.

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

Syntax:

```
@{ <Stmt>; }
```

Examples:

```
@{ int Count = 0; }  
@{ Count += 100; }  
@{ Object obj = new Object(); }  
@ { String str = "Hello World"; }
```

In-Line Statements: these statements are generally used for **accessing** or **printing** the 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 that can be **C#** or/and **HTML** also.

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 **C#** style of single line commenting i.e. “**//**”.

To test the above add, a new **Controller** in our current project i.e., “**MVCActionResults**”, name the Controller as “**RazorController**”, add a **View** to the default “**Index**” Action method, and write the below code in its “**<div>**” tag:

```
/* Single Line Statements */
{@{ int Count = 0; }
{@{ Count += 100; }
{@{ string str = "Hello World"; }
{@{ Object Obj = new Object(); }
```

```
/* In Line Statements */
Value of count is: @Count
<br>
Value of str is: @str
<br>
Obj is of type: @Obj.GetType()
<hr>
```

```

@* Multi Line Statements *@
@{
    string Date = DateTime.Now.ToString("yyyy-MM-dd");
    string Time = DateTime.Now.ToString("HH:mm:ss");
    <span>Today's Date is: @Date</span> <br>
    @:Current Time is: @Time <br>
    <text>Current Time is: @Time</text>
}
<hr>

```

```

@* If condition in Razor programming *@
@{
    int x = 131;
    if (x % 2 == 0)
    {
        <span>@x is an even number.</span>
    }
    else
    {
        <span>@x is an odd number.</span>
    }
}
<hr>

```

```

@{ string[] Colors = { "Red", "Blue", "Green", "Yellow", "Magenta" }; }
@* Loops in Razor programming - For Loop *@
@{
    <h4>List of colors printed using for loop:</h4>
    <ol>
        @for (int i = 0; i < Colors.Length; i++)
        {
            <li>@Colors[i]</li>
        }
    </ol>
}
<hr>

```

```

<h4>List of colors printed using for loop:</h4>
<ol>
    @for (int i = 0; i < Colors.Length; i++)
    {
        <li>@Colors[i]</li>
    }
</ol>
<hr>

```

```

<h4>List of colors printed using for loop:</h4>
<ol>
@{
    for (int i = 0; i < Colors.Length; i++)
    {
        <li>@Colors[i]</li>
    }
}
</ol>
<hr>

@* Loops in Razor programming - ForEach Loop *@
@{
    <h4>List of colors printed using foreach loop:</h4>
    <ul>
        @foreach (string color in Colors)
        {
            <li>@color</li>
        }
    </ul>
}
<hr>

<h4>List of colors printed using foreach loop:</h4>
<ul>
    @foreach (string color in Colors)
    {
        <li>@color</li>
    }
</ul>
<hr>

<h4>List of colors printed using foreach loop:</h4>
<ol>
@{
    foreach (string color in Colors)
    {
        <li>@color</li>
    }
}
</ol>

```

Passing values from Controller Action Methods to Views

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:

1. ViewData
2. ViewBag
3. TempData
4. Cookies
5. Session
6. Application
7. Anonymous Types
8. Models

To test all the above options, create a new “**ASP.Net Web Application**” project naming it as “**MVCDATATransfer**”, choose “**Empty Project Template**”, select “**MVC Checkbox**” and click on the **Create** button.

Now go to “RouteConfig.cs” file and modify the “url” value as below:

Old URL => url: "{controller}/{action}/{id}" New URL => url: "{controller}/{action}/{id}/{name}/{price}"

Now change the defaults also as following:

```
defaults: new
{
    controller = "Home",
    action = "Index",
    id = UrlParameter.Optional,
    name = UrlParameter.Optional,
    price = UrlParameter.Optional
}
```

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();

Add a new **Controller** in our project naming it as “**HomeController**”, delete the existing “**Index**” action method in the **class** and write the below code in the class:

```

public ViewResult Index1(int? id, string name, double? price)
{
    ViewData["Id"] = id;
    ViewData["Name"] = name;
    ViewData["Price"] = price;
    return View();
}

```

Now add a **View** to the “**Index1**” Action method (**with-out choosing any layout**) and write the following code inside of the “**<div>**” tag:

```

<table border="1" align="center" width="15%">
    <caption>Product Details</caption>
    <tr>
        <td>Id:</td><td>@ViewData["Id"]</td>
    </tr>
    <tr>
        <td>Name:</td><td>@ViewData["Name"]</td>
    </tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(ViewData["Price"]);
                Price = Price + (Price * 10 / 100);
                @Price
            }
        </td>
    </tr>
</table>

```

Execute the “**Index1**” Action method as following: <http://localhost:port/Home/Index1/101/Shoes/3500>, and in this case all the values that are sent to the **Action** method thru **URL** will be captured in **Action** method and stored into “**ViewData**” so that those values can be accessed under the corresponding **View** of **Index1**’s action method.

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 provide 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:

```
public ViewResult Display1()
{
    List<string> Colors = new List<string>() { "Red", "Blue", "Pink", "Black", "White", "Green", "Brown", "Purple" };
    ViewData["Colors"] = Colors;
    return View();
}
```

Now add a View to the “**Display1**” Action method (**with-out choosing any layout**) and write the following code inside of the “**<div>**” tag:

```
<h3>List of colors: </h3>
<ol>
    @foreach (string color in (List<String>)ViewData["Colors"])
    {
        <li>@color</li>
    }
</ol>
```

Execute “Display1” Action Method as following: <http://localhost:port/Home/Display1>

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-conversions** 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;

To test working with ViewBag add 2 new Action methods in the class Home Controller as below:

```
public ViewResult Index2(int? id, string name, double? price)
{
    ViewBag.Id = id;
    ViewBag.Name = name;
    ViewBag.Price = price;
    return View();
}
```

```

public ViewResult Display2()
{
    List<string> Colors = new List<string>() { "Red", "Blue", "Pink", "Black", "White", "Green", "Purple", "Yellow" };
    ViewBag.Colors = Colors;
    return View();
}

```

Now add a View to each action method and write the below code under their <div> tag:

Index2.cshtml:

```





```

Display2.cshtml:

```

<h3>List of colors: </h3>
<ol>
    @foreach (string color in ViewBag.Colors)
    {
        <li>@color</li>
    }
</ol>

```

Execute Index2 Action Method as following: <http://localhost:port/Home/Index2/105/Shoes/3500>

Execute Display2 Action Method as following: <http://localhost:port/Home/Display2>

As discussed above “ViewBag” is a wrapper around “ViewData” i.e., data that is stored in the “ViewBag” will be internally stored in “ ViewDataDictionary ” only, so values that are stored in a “ViewBag” can also be accessed thru “ViewData” and vice-versa. To test this, go to “Index2.cshtml” and change the statement `@ViewBag.Id` to `@ViewData["Id"]` and execute Index2 action method and still we get the same output as before.

Note: ViewBag is “Type Safe”, i.e., it doesn't require any type conversions while accessing data because we get the values back in their original types only, whereas ViewData requires type conversion because values are stored as object. ViewData and ViewBag are accessible only within the request i.e., with-in the corresponding View associated with Action Methods. In case of both if “Key Names” are misspelled then we don't get compile-time or runtime error.

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.

To test this, add 2 new action methods in “HomeController” class as below:

```
public RedirectToRouteResult Index3(int? id, string name, double? price)
{
    ViewData["Id"] = id;
    ViewBag.Name = name;
    TempData["Price"] = price;
    return RedirectToAction("Index4");
}

public ViewResult Index4()
{
    return View();
}
```

In the above case request will first come to “**Index3**” action method with a set of route values and that method stores those values into “**ViewData**”, “**ViewBag**” and “**TempData**” respectively, and then redirects to “**Index4**” action method which is performed by using “**RedirectToAction**” helper method and in this case return type of “**Index3**” action method should be “**RedirectToRouteResult**” but not “**ViewResult**”.

Now add a View to Index4 action method and write the below code in its “<div>” tag:

```
<table border="1" align="center" width="15%">
<caption>Product Details</caption>
<tr><td>Id:</td><td>@ViewData["Id"]</td></tr>
<tr><td>Name:</td><td>@ViewBag.Name</td></tr>
<tr>
    <td>Price:</td>
    <td>
        @{
            double Price = Convert.ToDouble(TempData["Price"]);
            Price = Price + (Price * 10 / 100);
        }
        @Price
    </td>
</tr>
</table>
```

Execute Index3 Action Method as following: <http://localhost:port/Home/Index3/105/Shoes/3500>

When the request is sent to “**Index3**” action method, it will redirect to “**Index4**” action method by calling “**RedirectToAction()**”, so this method will send a **re-direction response (302)** to the browser, so that browser will

send a new request to “Index4” action method and then “Index4” view gets launched and we can watch that in the browser’s address bar which will be shown as following: <http://localhost:port/Home/Index4>. If we watch the output in the browser, it will not display the values of “Id” and “Name” but will display the value of “Price” because “ViewData” and “ViewBag” can’t maintain the state of values between **multiple requests**, but “TempData” can maintain.

If we want to resolve the above problem, go to “Index3” action method and change “ViewData” and “ViewBag” to “TempData” and change the code in “Index4.cshtml” file and watch the difference.

In the above example by using “RedirectToAction” helper method we are able to transfer the control from 1 **action** method to another **action** method that is present in the same **controller**, whereas we can also transfer to an **action** method that is present in another **controller** also, and to test it add a new **action** method in the “HomeController” class as below:

```
public RedirectToRouteResult Index5(int? id, string name, double? price)
{
    TempData["Id"] = id;
    TempData["Name"] = name;
    TempData["Price"] = price;
    return RedirectToAction("Index1", "Test");
}
```

In the above code under “RedirectToAction” helper method, “Test” is the **controller’s** name and “Index1” is the **action** method name where it must redirect. **RedirectToAction** helper method is an overloaded method that is pre-defined as following:

```
RedirectToAction(string actionPerformed)
RedirectToAction(string actionPerformed, string controllerName)
```

Now add a new **Controller** in the project naming it as “**TestController**”, change the default **Action** method name i.e., “**Index**” to “**Index1**”, add a **View** to the method and write the below code in the **View** under “**<div>**” tag:

```
<table border="1" align="center" width="15%">
<caption>Product Details</caption>
<tr><td>Id:</td><td>@ TempData["Id"]</td></tr>
<tr><td>Name:</td><td>@ TempData["Name"]</td></tr>
<tr>
    <td>Price:</td>
    <td>
        @{
            double Price = Convert.ToDouble(TempData["Price"]);
            Price = Price + (Price * 10 / 100);
            @Price
        }
    </td>
</tr>
</table>
```

Execute Index5 Action Method of Home Controller as following:

<http://localhost:port/Home/Index5/105/Shoes/3500>

After execution of Index5 Action method, the URL in address bar will be as following because it is re-directed:

<http://localhost:port/Test/Index1>

Flow of control: Request sent to => **Index5** action method of **HomeController** => Redirected to => **Index1** action method of **TestController**.

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**.

Drawback of TempData: If we access values once from “**TempData**” immediately all those values get deleted and can't be accessed in next requests, to test this add a hyper link after the table in “**Index1.cshtml**” file of “**TestController**” as following:

```
<center>Click to launch new <a href="/Test/Display1">Page</a>.</center>
```

So, if we click on the hyper link it must call “**Display1**” action method of “**TestController**” class, so add a new action method in “**TestController**” class as following:

```
public ViewResult Display1()
{
    return View();
}
```

Add a View to Display1 action method and write the below code in its “<div>” tag:

```
<table border="1" align="center" width="15%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@ TempData["Id"]</td></tr>
    <tr><td>Name:</td><td>@ TempData["Name"]</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(TempData["Price"]);
                Price = Price + (Price * 10 / 100);
                @Price
            }
        </td>
    </tr>
</table>
```

Execute Index5 Action Method of HomeController as following:

<http://localhost:port/Home/Index5/105/Shoes/3500>



This will launch “Index1” view of “TestController” displaying the values and now click on the **hyper link** we added to launch “Display1” view of “TestController”, but “Display1” view will not display the values because we accessed values in “Index1” view and immediately those values got deleted, so can’t be accessed in “Display1” view again.

Flow of control: Request sent to => **Index5** action method of **HomeController** => Redirected to => **Index1** action method of **TestController** => when the link is clicked redirects to => **Display1** Action method of **TestController**.

To **overcome** the above problem and access the values in next requests, we need to use the “**Peek**” method of “**TempData**” while accessing values, so that the values will be persisting even after accessing them and can be accessed again. To do that, change the code of accessing data in “**Index1**” view as following:

```
@ TempData["Id"]      =>      @ TempData.Peek("Id")
@ TempData["Name"]    =>      @ TempData.Peek("Name")
@ TempData["Price"]   =>      @ TempData.Peek("Price")
```

Note: In this case if we want to transfer data from “**Display1**” view to any other views there also we must change the code as above.

To retain the values of “**TempData**” after accessing, we can also call “**Keep**” method in place of “**Peek**”, but the difference is when we use “**Keep**” it will retain **all** the values of “**TempData**” and can be consumed in the next request, where “**Peek**” method is used for retaining **specific** values only. So, without calling the **Peek** method 3 times we can simply call the **Keep** method and to try this write the below code, above the “**<table>**” tag in “**Display1.cshtml**” file:

```
@{ TempData.Keep(); }
```

Note: Call **Keep** for **preserve** all the values of **TempData** and call **Peek** to **preserve** specific values of **TempData**.

Cookie: 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 re-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 Site** or **Web Application** and sends them to the **Server**. Browser stores **cookies** separately for each different **site** we visit.

Writing Cookies on client machine: To write a **Cookie** on **Client Machine** first we need to create the **instance** of **HttpCookie** class; store values into it treating it like a **dictionary** and then write that **Cookie** to **Client Machine** by using **Response** object.

```
HttpCookie cookie = new HttpCookie("LoginCookie");
cookie["User"] = "Raju";           cookie["Pwd"] = "Admin@123";
Response.Cookies.Add(cookie);
```

Reading Cookies back on our WebPages: we can read **Cookies Present** on the **Client Machine** from a **View** or **Action Method** using **Request** object.

```
HttpCookie cookie = Request.Cookies["LoginCookie"];
string User = cookie["User"];
string Pwd = cookie["Pwd"];
```

Cookies are of 2 types:

- I. In-Memory Cookies
- II. 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.

Setting expires property of Cookie:

```
cookie.Expires = <DateTime>;
```

To test working with **Cookies**, add 2 new Action methods in "**HomeController**" for creating a **Cookie** with a set of values and then launching the View to display those values as below:

```
public ViewResult Index6(int? Id, string Name, double? Price)
{
    HttpCookie cookie = new HttpCookie("ProductCookie");
    cookie["Id"] = Id.ToString();
    cookie["Name"] = Name;
    cookie["Price"] = Price.ToString();
    cookie.Expires = DateTime.Now.AddDays(3);
    Response.Cookies.Add(cookie);
    return View();
}
public ViewResult Index7()
{
    return View();
}
```

Now add a View to "Index6" Action method and write the below code under its "<div>" tag:

```
@{
    HttpCookie cookie = Request.Cookies["ProductCookie"];
    int Id = int.Parse(cookie["Id"]);
    string Name = cookie["Name"];
    double Price = double.Parse(cookie["Price"]);
}






```

```

    Price = Price + Price * 10 / 100;
    @Price
}
</td>
</tr>
</table>
<center>Click to launch new <a href="/Home/Index7">Page</a>.</center>

```

Execute Index6 Action Method of Home Controller as following:

<http://localhost:port/Home/Index6/105/Shoes/3500>

In the above case when we call **Index6** action method, will launch the associated **view** and displays the values that are stored in the **cookie** and when we click on the **Link** in the bottom will call **Index7** action method of the same **controller** which will also display the values that stored in the **cookie** and to test that add a **View** for **Index7** action method and write the below code in it under the “**<div>**” tag:

```

{@{
HttpCookie cookie = Request.Cookies["ProductCookie"];
int Id = int.Parse(cookie["Id"]);
string Name = cookie["Name"];
double Price = double.Parse(cookie["Price"]);
}

<table border="1" align="center" width="20%">
<caption>Product Details</caption>
<tr><td>Id:</td><td>@Id</td></tr>
<tr><td>Name:</td><td>@Name</td></tr>
<tr>
<td>Price:</td>
<td>
@{
    Price = Price + Price * 10 / 100;
    @Price
}
</td>
</tr>
</table>
<center>Click to launch new <a href="/Test/Index2">Page</a>.</center>

```

When we click on the **Link** in above **View** will launch **Index2** action method of **TestController** and here also it displays the values that are stored in **Cookie** and to test it add a new action method in **TestController** class as below:

```

public ViewResult Index2()
{
    return View();
}

```

Add a View to the above Index2 action method and write the below code under it's "<div>" tag:

```
@{  
    HttpCookie cookie = Request.Cookies["ProductCookie"];  
    int Id = int.Parse(cookie["Id"]);  
    string Name = cookie["Name"];  
    double Price = double.Parse(cookie["Price"]);  
}  
  
<table border="1" align="center" width="20%">  
    <caption>Product Details</caption>  
    <tr><td>Id:</td><td>@Id</td></tr>  
    <tr><td>Name:</td><td>@Name</td></tr>  
    <tr>  
        <td>Price:</td>  
        <td>  
            @{  
                Price = Price + Price * 10 / 100;  
                @Price  
            }  
        </td>  
    </tr>  
</table>
```

Flow of control: Request sent to => “Index6” Action method of “HomeController” => when the hyper link is clicked on page, it is redirect to => “Index7” Action method of “HomeController” => when the hyper link is clicked on page, and it is redirect to => “Index2” Action method of “TestController”.

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**.

Location of Persistent Cookies:

Microsoft Edge: C:\Users\<User>\AppData\Local\Microsoft\Edge\User Data\Default

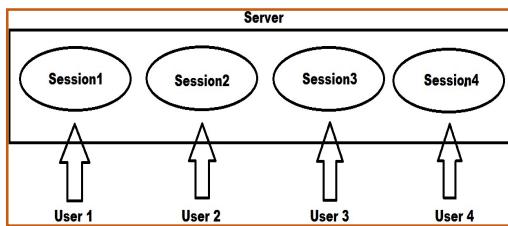
Google Chrome: C:\Users\<User>\AppData\Local\Google\Chrome\User Data\Default

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”. We can also delete individual cookies using “Manage and delete cookies and site data” option.

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 another `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`”.



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();`

To test `Sessions`, add 2 new Action methods in “`HomeController`” as below for transferring `session` values from `Action` method to `Action` method within a `Controller`:

```

public RedirectToRouteResult Index8(int? id, string name, double? price)
{
    Session["Id"] = id;
    Session["Name"] = name;
    Session["Price"] = price;
    return RedirectToAction("Index9");
}

public ViewResult Index9()
{
    return View();
}
  
```

In the above case we are storing the values into a `Session` in “`Index8`” action method and then redirecting to “`Index9`” action method of same `Controller`, so the `View` that is associated with “`Index9`” can access the values from the `Session` and display them. Now add a `View` to “`Index9`” Action method and write the below code under its “`<div>`” tag:

```

<table border="1" align="center" width="15%>
<caption>Product Details</caption>
<tr><td>Id: </td><td>@Session["Id"]</td></tr>
<tr><td>Name: </td><td>@Session["Name"]</td></tr>
<tr>
<td>Price: </td>
<td>
<{ 
    double Price = Convert.ToDouble(Session["Price"]);
    Price = Price + (Price * 10 / 100);
    @Price;
}
</td>
</tr>
</table>

```

Execute Index8 Action Method of Home Controller as following:

<http://localhost:port/Home/Index8/105/Shoes/3500>

After execution of Index8 Action Method, the URL in the address bar will be as following because it re-directed:

<http://localhost:port/Home/Index9>

Flow of control: Request sent to => “Index8” Action method of “HomeController” => Redirected to => “Index9” Action method of “HomeController”.

Values that are stored in a **Session** are accessible in **Action** methods or **Views** which are defined in same or different **Controller** also and for testing this add a new **Action** method in “HomeController” as below:

```

public RedirectToRouteResult Index10(int? id, string name, double? price)
{
    Session["Id"] = id;
    Session["Name"] = name;
    Session["Price"] = price;
    return RedirectToAction("Index3", "Test");
}

```

In the above case “Index10” action method of “HomeController” is storing the values into a **Session** and then redirecting to “**TestController**” => “Index3” action method, so the view that is associated with the “Index3” action method can access the values from **Session** and from there if we further redirect to any other **View** also, we can access the values from the **Session** and to test that add 2 new **Action** methods in “**TestController**” as below:

```

public ViewResult Index3()
{
    return View();
}

public ViewResult Display2()

```

```
{
    return View();
}
```

Add a View to “Index3” Action method and write the below code under its “<div>” tag:

```
<table border="1" align="center" width="15%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@Session["Id"]</td></tr>
    <tr><td>Name:</td><td>@Session["Name"]</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(Session["Price"]);
                Price = Price + (Price * 10 / 100);
                @Price
            }
        </td>
    </tr>
</table>
<center>Click to launch new <a href="/Test/Display2">Page</a>.</center>
```

Add a View to Display2 Action method and write the below code under its “<div>” tag:

```
<table border="1" align="center" width="15%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@Session["Id"]</td></tr>
    <tr><td>Name:</td><td>@Session["Name"]</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(Session["Price"]);
                Price = Price + (Price * 10 / 100);
                @Price
            }
        </td>
    </tr>
</table>
```

Execute Index10 Action Method of Home Controller as following:

<http://localhost:port/Home/Index10/105/Shoes/3500>

Flow of control: Request sent to => “Index10” Action method of “HomeController” => Redirected to => “Index3” Action method of “TestController” => when the hyper link is clicked on this page, it is again redirected to => “Display2” Action method of “TestController” and provides access to all the values that are stored in the Session but for this User only.

How is a session identified to which user it belongs to?

Ans: Whenever a Session is created for a user it is given with a Unique ID known as “**SessionId**” and this “**SessionId**” is written to client’s browser in the form of a “**In-Memory Cookie**”, so whenever the client comes back to the server in the next **request**, server will read the **Cookie**, picks the “**SessionId**” and associates user with his exact **Session**.

Note: because **SessionId** is stored on client’s browser in the form of an “**In-Memory Cookie**”, other tabs under the browsers **instance** can also access that **Session**, whereas a new **instance** of a new **browser** can’t access the **Session**.

What happens to Sessions associated with clients if the client closes the browser?

Ans: Every **Session** will be having a “**time-out**” period of “**20 Minutes (default)**” from the last request (**Sliding Expiration**), so within **20 Minutes** if the **Session** is not used by the **User**, **Server** will destroy that, **Session**.

Note: we can change the default **time-out** period of “**20 Mins**” to our required value thru “**Web.config**” file by setting “**timeout**” attribute value of “**sessionState**” element. To test it write the below code under “**<system.web>**”:

```
<sessionState timeout="1" />
```

Can we explicitly destroy a Session associated with a User?

Ans: Yes, this can be performed by calling “**Abandon()**” method on the **Session** and this is what we generally do under “**Sign Out**” or “**Log Out**” options in a **Web Site** or **Web Application**.

E.g.: `Session.Abandon();`

Where will Web Server store Session values?

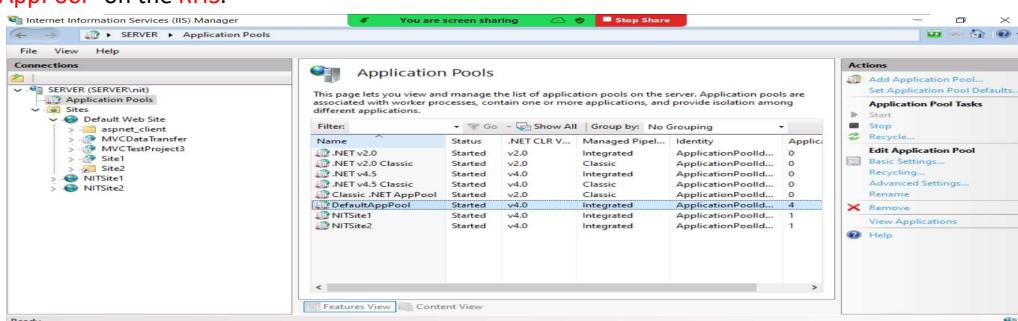
Ans: Web Server can store **Session** values in 3 different locations:

- I. In-Proc [d]
- II. State Server
- III. SQL Server

In-Proc: this is the **default** option used for storing **Session** values and in this case **Session** values are stored under the memory of “**IIS Worker Process**”. To understand about these, first host your application on “**Local IIS**”.

What is IIS Worker Process?

Ans: Under **IIS**, Web Application runs inside of a container known as **Application Pool** or an **Application Pool** is a container of **Web Applications** that will execute by a single or multiple **Worker Process**. **Application Pool** is the **Heart** of a **Web Application** and by default under **IIS** all **Web Applications** runs under the same **Application Pool** which is created when **IIS** is installed i.e., “**Default App Pool**”. To check that open “**IIS Manager**” and in the **LHS** under “**Connections Panel**” we will find “**Applications Pools**” and “**Sites**” options, select “**Application Pools**” which displays “**DefaultAppPool**” on the **RHS**.

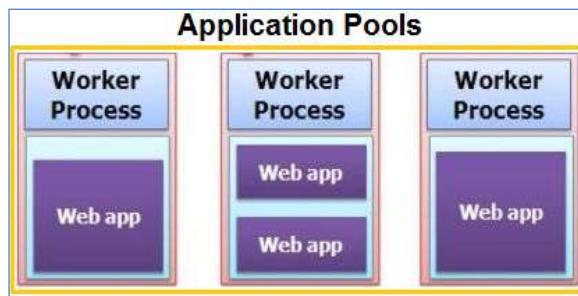


If you notice the above image there are **4 applications** on the server running under “**DefaultAppPool**” and it is still possible to run each **Web Application** under a separate **Application Pool** which enables us to isolate our **Web Application** for better **Security, Reliability and Availability**; therefore, problems in one **Application Pool** do not affect **Web Sites or Applications** in other Application Pools.

Note: whenever a new **Site** is created under **IIS**, it will also create an **Application Pool** under which the new **Site** runs and name of that **Application Pool** will be the same name of **Site**. In the above picture you can see **NITSite1** and **NITSite2** Application Pools associated with **NITSite1** and **NITSite** Site’s

We can also create our own **Application Pools** under **IIS** and to do that right click on **Application Pools** node under **Connection Panel** and select “**Add Application Pool**” which opens a window asking for a name, enter name as “**MyPool**” and click “**Ok**”. Currently the new **Application Pool** i.e., “**MyPool**” doesn’t have any applications running under it and if we want our application “**MVCDataTransfer**” to run under “**MyPool**”, then right click on our application under “**Default Web Site**” and choose **Manage Applications => Advanced Settings**, which opens a window and in that select “**Application Pool**” and click on the button beside it which opens another window listing all the **Application Pools** that are available in a **DropDownList**, select “**MyPool**” and click **Ok** and **Ok** again.

The **IIS Worker Process** is a **Windows Process** (**w3wp.exe**) which runs **Web Applications** and is responsible for handling requests sent to a **Web Server** for a specific **Application Pool**. Each **Application Pool** creates at least **1 instance** of **w3wp.exe** and that is responsible for processing **ASP.NET** application **request** and sending back **response** to the **client**. All **ASP.NET** functionalities run within the scope of this **Worker Process**.



We can view the “**IIS Worker Process**” that is associated with each **Application Pool** under “**Task Manager**” which displays a separate “**IIS Worker Process**” for each **Application Pool**. To get the exact details go to “**Details Section**” in the **Task Manager** and there we find “**w3wp.exe**” and beside that it will display the **Application Pool** to which this **Worker Process** is associated.

Note: in **In-Proc** session mode **Session values** are stored under the **Memory** that is associated with **IIS Worker Process**, who runs our **Web Application**. So, in this case if we **recycle** the **IIS Worker Process** all the **Session Values** that are stored under this will be destroyed and to test this, open “**Task Manager**” identify the “**IIS Worker Process**” under which our **Web Application** is running, select it and click on “**End Task**” button which will destroy all the **session values** associated with that application.

State Server: this is separate software for storing “**Session Values**” which is installed when we install **.NET Runtime** on any machine, and it can be found in the “**Services Window**”. To see that go to **Control Panel => Administrative Tools => Services** and in that we find “**ASP.NET State Service**”. To use this, first we need to set “**mode**” attribute of **<sessionState>** element in the “**Web.config**” file and “**mode**” attribute accepts any of the following values like: **Off**, **In-Proc [d]**, **SQL Server** and **State Server**.

Note: default is In-Proc i.e., Sessions are stored under IIS Worker Process Memory, if set as off application will not maintain Sessions at all and if set as State Server then Session values are stored under ASP.NET State Service.

To use State Service for storing Session values we need to do the following:

Step 1: Open Windows Services console, right click on ASP.NET State Service and select Start to start the service.

Step 2: Now open Web.config file and write the sessionState tag as following:

```
<sessionState mode="StateServer" stateConnectionString="tcpip=localhost:42424" />
```

Note: “localhost” refers to the machine name and if ASP.NET State Service software is running on a remote machine then write that machine name in place of localhost and 42424 is the Port No. on which ASP.NET State Server software will be running.

SQL Server: if “Session Mode” is set as SQL Server then Session values are stored under SQL Server Database and to use that we need to create a Database, a set of Tables and Stored Procedures. Without creating all these objects manually, we are provided with a command line tool called as “aspnet_regsql” that should be used from “Visual Studio Developer Command Prompt” as below:

Step 1: run the “aspnet_regsql” tool at “VS Developer Command Prompt”, so that the required Database, Tables and Stored Procedure for maintaining Sessions will be created under SQL Server.

```
aspnet_regsql -? => Help  
aspnet_regsql -S <Server Name> -U <User Id> -P <Password> -E <In-case of Windows Auth> -ssadd -sstype t|p|c
```

-S: to specify SQL Server name.

-U: to specify User Id in case of SQL Authentication.

-P: to specify password in case of SQL Authentication.

-E: this must be used in case of Windows Authentication and in such case don't use -U and -P option again.

-ssadd: is to enable support for SQL Server Session State and this will create a Database on the server.

Note: If we want to remove the support for session state we need to use -ssremove in place -ssadd.

-sstype: is to specify the type of tables we want to use where “t” indicates temporary tables, “p” indicates persisted tables and “c” indicates custom - but in this case we need to create our own Database to store Session data, and to specify that Database name we need to use “-d <Database Name>” option in the last.

Testing the processes of using “aspnet_regsql”:

For SQL Server Authentication:	aspnet_regsql -S Server -U Sa -P 123 -ssadd -sstype t
For Windows Authentication:	aspnet_regsql -S Server -E -ssadd -sstype t

Note: in the above case we are using temporary tables for storing Session State values, so a new Database is created on the Server with the name “ASPState” and under this Database it creates a set of Stored Procedures for managing the data and because we have asked for temporary tables, all the required tables gets created on “TempDB - System Database” and this Database will be re-created every time we re-start SQL Server, whereas if we ask for persisted tables then all the required tables also gets created on “ASPState” Database only, so even if we re-start SQL Server, then also tables and their values will be persisting.

Step 2: Now open Web.config file and re-write the `<sessionState>` tag as below:

Sql Server Authentication:

```
<sessionState mode="SQLServer" sqlConnectionString="Data Source=Server;User Id=Sa;Password=123" />
```

Windows Authentication:

```
<sessionState mode="SQLServer" sqlConnectionString="Data Source=Server;Integrated Security=SSPI" />
```

Testing the processes of using “aspnet_Regsql” with permanent table support:

For SQL Server Authentication: aspnet_Regsql -S Server -U Sa -P 123 -ssadd -sstype p

For Windows Authentication: aspnet_Regsql -S Server -E -ssadd -sstype p

Testing the processes of using “aspnet_Regsql” with custom Database and permanent table support:

For SQL Server Authentication: aspnet_Regsql -S Server -U Sa -P 123 -ssadd -sstype c -d MySessions

For Windows Authentication: aspnet_Regsql -S Server -E -ssadd -sstype c -d MySessions

In this case we are using **Custom Database** then we need to specify the **Database** name in our `sqlConnectionString` and also we need to use `allowCustomSqlDatabase` attribute with the value as **true** as below:

Sql Server Authentication:

```
<sessionState mode="SQLServer" allowCustomSqlDatabase="true"  
sqlConnectionString="Data Source=Server;User Id=Sa;Password=123;Database=MySessions" />
```

Windows Authentication:

```
<sessionState mode="SQLServer" allowCustomSqlDatabase="true"  
sqlConnectionString="Data Source=Server; Integrated Security=SSPI;Database=MySessions" />
```

Removing support for Sql Server Session State:

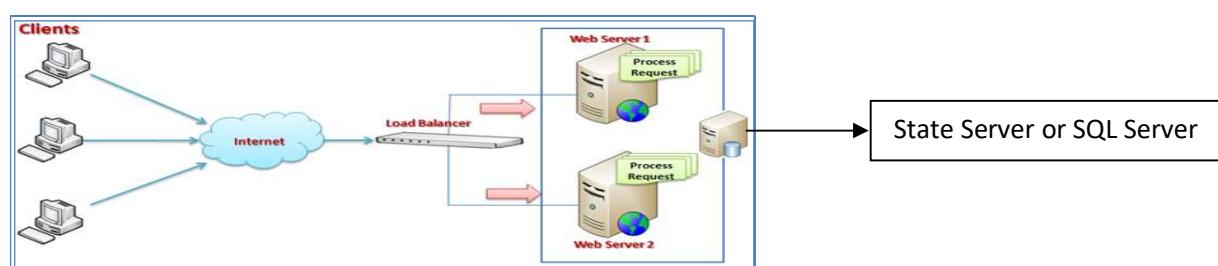
For SQL Server Authentication: aspnet_Regsql -S Server -U sa -P 123 -ssremove

For Windows Authentication: aspnet_Regsql -S Server -E -ssremove

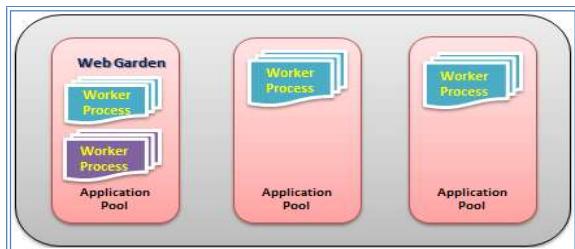
When to use State Server and SQL Server Session Modes over In-Proc Session Mode?

Ans: State Server and SQL Server session mode options are used in scenarios where a **Web Application** is running in “**Web Farm**” or “**Web Garden**” architectures.

Web Farm: it's an approach of hosting a **Web Application** on multiple **Web Servers** for balancing the load, so that client request first comes to **Load Balancer** and **Load Balancer** will in-turn redirect the client to an appropriate **Web Server** which is free currently. In case of **Web Farm** architecture “**In-Proc**” Session Mode can't be used because if each request from the same client is re-directed to a different **Web Servers**, then **Session Data** of 1 **Web Server** is not accessible to other **Web Servers**, so to overcome this problem we install “**State Server**” or “**SQL Server**” software on a **remote machine** and configure all the **Web Servers** to that machine as following:

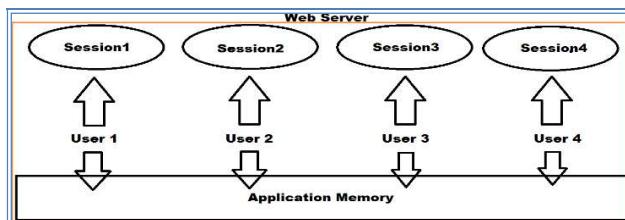


Web Garden: as discussed earlier every **Application Pool** under which our **Web Application** executes will be having **1 Worker Process** to execute that **Application**, but we can have more than **1 Worker Process** also in an **Application Pool** and if we do that, we call that as **Web Garden**.



To add **Worker Processes** to an **Application Pool** right click on the **Application Pool** in “IIS Manager”, select “Advanced Settings” which opens a window and, in that window, under the “Process Model” settings we find “Maximum Worker Processes” with the value **1**, change it to any value other than **1** and click **Ok**.

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 memory 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**” namespace.

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: `HttpContext.Application[string key] = value (object)`
Example: `HttpContext.Application["Name"] = "Raju";`

Accessing values from Application:

Syntax: `object value = HttpContext.Application[string key]`
Example: `object value = HttpContext.Application["Name"];
string Name = value.ToString();`
 Or
`string Name = HttpContext.Application["Name"].ToString();`

```

public ViewResult Index11(int? id, string name, double? price)
{
    HttpContext.Application.Lock();
    HttpContext.Application["Id"] = id;
    HttpContext.Application["Name"] = name;
    HttpContext.Application["Price"] = price;
    HttpContext.Application.UnLock();
    return View();
}
public ViewResult Index12()
{
    return View();
}

```

Now add a View to “Index11” Action method and write the below code under its “<div>” tag:

```

<table border="1" align="center" width="20%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@HttpContext.Current.Application["Id"]</td></tr>
    <tr><td>Name:</td><td>@HttpContext.Current.Application["Name"]</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(HttpContext.Current.Application["Price"]);
                Price = Price + Price * 10 / 100;
            }
            @Price
        </td>
    </tr>
</table>
<center>Click to launch new <a href="/MVCDATATransfer/Home/Index12">Page</a>.</center>

```

Execute Index11 Action Method of Home Controller as following:

<http://localhost/MVCDATATransfer/Home/Index11/105/Shoes/3500>

In the above case when we call **Index11** action method, will launch the associated **View** and displays the values that are stored in the **Application Memory** and when we click on the **Link** in the bottom will call **Index12** action method of the same **controller** which will also display the values that stored in **Application Memory** and to test that add a **View** for **Index12** action method and write the below code in it under the “**<div>**” tag:

```

<table border="1" align="center" width="20%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@HttpContext.Current.Application["Id"]</td></tr>
    <tr><td>Name:</td><td>@HttpContext.Current.Application["Name"]</td></tr>
    <tr>
        <td>Price:</td>

```

```

<td>
    @{
        double Price = Convert.ToDouble(HttpContext.Current.Application["Price"]);
        Price = Price + Price * 10 / 100;
    }
</td>
</tr>
</table>
<center>Click to launch new <a href="/MVCDataTransfer/Test/Index4">Page</a>.</center>

```

When we click on the **Link** in above **View** will launch **Index4** action method of **TestController** and here also it displays the values that are stored in **Cookie** and to test it add a new action method in **TestController** class as below:

```

public ViewResult Index4()
{
    return View();
}

```

Add a View to the above Index4 action method and write the below code under it's "<div>" tag:

```

<table border="1" align="center" width="20%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@HttpContext.Current.Application["Id"]</td></tr>
    <tr><td>Name:</td><td>@HttpContext.Current.Application["Name"]</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double Price = Convert.ToDouble(HttpContext.Current.Application["Price"]);
                Price = Price + Price * 10 / 100;
            }
        </td>
    </tr>
</table>

```

Flow of control: Request sent to => **"Index11"** Action method of **"HomeController"** => when the hyper link is clicked on page, it is redirect to => **"Index12"** Action method of **"HomeController"** => when the hyper link is clicked on page, and it is redirect to => **"Index4"** Action method of **"TestController"**.

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.

For example if we define a class as below:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Job { get; set; }
    public double Salary { get; set; }
    public bool Status { get; set; }
}
```

We can create instance of the above class as following:

```
Employee Emp = new Employee { Id = 1001, Name = "Raju", Job = "Manager", Salary = 50000.00, Status = true };
```

In the above case we are creating the instance of “Employee” class which is defined first and then initializing the attributes (**Id**, **Name**, **Job**, **Salary**, and **Status**) that are associated with it, so now the instance “**Emp**” contains all attribute values in it. Without **explicitly** defining a **class/type** also, we can create an instance for a type and initialize it as following:

```
var Emp = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 50000.00, Status = true };
dynamic Emp = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 50000.00, Status = true };
```

In the above case also instance “**Emp**” is an instance containing all the attribute values in it same as in the first scenario, but the difference is in the first case we defined the class first and then created the instance whereas in the second case we created the instance without defining the class/type and we call this “**Anonymous Type**”.

Anonymous type (**introduced in C# 3.0**), as the name suggests, is a type that doesn’t have any name. C# allows you to create an instance with the **new** keyword without defining a class. The implicitly typed variable - “**var**” or “**dynamic**” is used to hold the **reference** of anonymous types. In the above example, “**Emp**” is an instance of the anonymous type which is created by using the **new** keyword and object initializer syntax. It includes 5 properties of different data types. An anonymous type is a **temporary type** that is inferred based on the data that you include in an object initializer. **Properties** of anonymous types will be **read-only** properties so you cannot change their values.

Notice that the compiler applies the appropriate type to each property based on the value assigned. For example, **Id** is of integer type, **Name** and **Job** are of string type, **Salary** is of double type and **Status** is of Boolean type. Internally, the compiler automatically generates the new type for anonymous types. You can check that by calling **GetType()** method on an anonymous type of instance which will return the following value:

```
<>f__AnonymousType0`5[System.Int32,System.String,System.String,System.Double,System.Boolean]
```

By using these anonymous types also, we can transfer values from an **Action** method to its corresponding **View** or an **Action** method to another **Action** method that is present in the same **Controller** or other **Controller**.

Define a new Action method in Home Controller as below:

```
public ViewResult Index13(int? id, string name, double? price)
{
    var Product = new { Id = id, Name = name, Price = price };
    return View(Product);
}
```

Add a View to the above Action method and write the below code under its “<div>” tag:

```
@{  
    int? Id = Convert.ToInt32(ViewData.Eval("Id"));  
    string Name = ViewData.Eval("Name").ToString();  
    double? Price = ViewData.Eval("Price") as double?;  
}  
  
<table border="1" align="center" width="15%">  
    <caption>Product Details</caption>  
    <tr><td>Id:</td><td>@Id</td></tr>  
    <tr><td>Name:</td><td>@Name</td></tr>  
    <tr>  
        <td>Price:</td>  
        <td>  
            @{  
                Price = Price + (Price * 10 / 100);  
                @Price  
            }  
        </td>  
    </tr>  
</table>
```

In the above case while accessing values from an **anonymous type** in the **View**, we need to first capture those values from **ViewData** i.e., even if we did not **explicitly** store the values in **ViewData**, they are **implicitly** stored into **ViewData** and accessible under the **View**.

While accessing the values from **ViewData** we require calling **“Eval”** method on **ViewData**, because the values are stored in the form of an **“instance or object (Complex type)”** and under **that instance or object** we have all the 3 values which should be separated first and then consumed.

Execute Index13 Action Method of Home Controller as following:

<http://localhost/MVCDataTransfer/Home/Index13/105/Shoes/3500>

Now define another new Action method in Home Controller as below:

```
public RedirectToRouteResult Index14(int? id, string name, double? price)  
{  
    var product = new { Id = id, Name = name, Price = price };  
    return RedirectToAction("Index5", "Test", product);  
}
```

Define a new Action method in “TestController” as below:

```
public ViewResult Index5(int? id, string name, double? price)  
{  
    var product = new { Id = id, Name = name, Price = price };  
    return View(product);  
}
```

Add a View to Index5 Action method and write the below code in its “<div>” tag:

```
@{  
    int? Id = Convert.ToInt32(ViewData.Eval("Id"));  
    string Name = ViewData.Eval("Name").ToString();  
    double? Price = ViewData.Eval("Price") as double?;  
}  
  
<table border="1" align="center" width="15%">  
    <caption>Product Details</caption>  
    <tr><td>Id:</td><td>@Id</td></tr>  
    <tr><td>Name:</td><td>@Name</td></tr>  
    <tr>  
        <td>Price:</td>  
        <td>  
            @{@  
                Price = Price + (Price * 10 / 100);  
                @Price  
            }  
        </td>  
    </tr>  
</table>
```

Execute Index14 Action Method of Home Controller as following:

<http://localhost/MVCDATATransfer/Home/Index14/105/Shoes/3500>

Now URL in the address bar will change as following:

<http://localhost/MVCDATATransfer/Test/Index5/105/Shoes/3500>

Flow of control: Request sent to => Index14 action method of HomeController => Redirected to => Index5 action method of TestController and in this process all the 3 values are sent to Index5 action method as Route Parameters, as shown above.

Model Object: this is another way how we can transfer data from Controller to the View, which provides “type safety” and “intellisense” support in Views, so by using this also we can transfer values from Action method to corresponding View or Action method to Action method of same Controller or other Controllers. To test this, first add a class in the project under “Models” folder naming the class as “Product” and write the below code in it:

```
public class Product  
{  
    public int? Id { get; set; }  
    public string Name { get; set; }  
    public double? Price { get; set; }  
}
```

Now import the namespace “MVCDATATransfer.Models” in “HomeController” class and define a new action method as below:

```

public ViewResult Index15(int? id, string name, double? price)
{
    Product product = new Product { Id = id, Name = name, Price = price };
    return View(product);
}

```

In the above action method, we can write code in any of these ways:

```

Product product = new Product { Id = id, Name = name, Price = price };
return View(product);

```

Or

```

var product = new Product { Id = id, Name = name, Price = price };
return View(product);

```

Or

```

dynamic product = new Product { Id = id, Name = name, Price = price };
return View(product);

```

Or

```

return View(new Product { Id = id, Name = name, Price = price });

```

Add a View to “Index15” Action method and write the below code:

```

@*Write the below statement at top of the View *@
@model MVCDataTransfer.Models.Product

```

```

@*Write this code under the <div> tag*@
<table border="1" align="center" width="15%">
    <caption>Product Details</caption>
    <tr><td>Id:</td><td>@Model.Id</td></tr>
    <tr><td>Name:</td><td>@Model.Name</td></tr>
    <tr>
        <td>Price:</td>
        <td>
            @{
                double? Price = Model.Price + (Model.Price * 10 / 100);
                @Price
            }
        </td>
    </tr>
</table>

```

Note: in this case on top of the **View** we need to first import the model object by using “**@model**” directive, so that we can access the member of the **Product** class by using “**@Model**” Property.

Execute “Index15” Action Method as following:

<http://localhost/MVCDataTransfer/Home/Index15/105/Shoes/3500>

In the above “**Index15**” action method without reading each parameter value individually we can directly read it thru the **Model** class, by using the class as method parameter.

To test that let's now add another Action method in "HomeController" class as following:

```
public RedirectToRouteResult Index16(Product product)
{
    return RedirectToAction("Index6", "Test", product);
}
```

Add a new Action method in "Test Controller" class as following:

```
public ViewResult Index6(Product product)
{
    return View(product);
}
```

Note: now add a **View** to **Index6 Action** method of **Test Controller** and write code same as we written in **Index15 Action** method of **Home Controller**.

Execute Index16 Action Method as following:

<http://localhost:MVCDataTransfer/Home/Index16/105/Shoes/3500>

Now URL in the address bar will change as following:

<http://localhost/MVCDataTransfer/Test/Index6/105/Shoes/3500>

Flow of control: Request sent to => **Index16** action method of **HomeController** => Redirected to => **Index6** action method of **TestController** and in this process all the 3 values are sent to **Index6** action method as **Route Parameters**, as shown above.

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 may 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 design them by using any of the below options:

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

To test these, create a new “**ASP.NET Web Application**” project naming it as “**MVCUIDesigning**”, choose “**Empty Project Template**”, check the “**MVC**” Checkbox and click on “**Create**” button to create the project.

Designing a View (UI) by using HTML Controls: Add a **Controller** into the **Controllers** folder naming it as “**EmpController**”, delete all the existing code in the class and write the below code:

```
[HttpGet]
public ViewResult AddEmp()
{
    return View();
}

[HttpPost]
public ViewResult AddEmp(int? id, string name, string job, double? salary)
{
    ViewData["Id"] = id;
    ViewData["Name"] = name;
    ViewData["Job"] = job;
    ViewData["Salary"] = salary;
    return View("DisplayEmp1");
}
```

Add a view to “AddEmp” action method and write the below code in its “<div>” tag:

```
<form method="post">
    Enter Employee Id: <br />
    <input type="text" name="Id" id="Id" /><br />
    Enter Employee Name: <br />
    <input type="text" name="Name" id="Name" /><br />
    Enter Employee Job: <br />
    <input type="text" name="Job" id="Job" /><br />
    Enter Employee Salary: <br />
    <input type="text" name="Salary" id="Salary" /><br /><br />
    <input type="submit" value="Save" />
    <input type="reset" value="Reset" />
</form>
```

Add another view in “**Emp**” folder of “**Views**” folder with the name “**DisplayEmp1**” and write the below code in its “**<div>**” tag:

```

<table border="1" align="center">
  <caption>Employee Details</caption>
  <tr><td>Employee Id:</td><td>@ViewBag.Id</td></tr>
  <tr><td>Employee Name:</td><td>@ViewBag.Name</td></tr>
  <tr><td>Employee Job:</td><td>@ViewBag.Job</td></tr>
  <tr><td>Employee Salary:</td><td>@ViewBag.Salary</td></tr>
  <tr>
    <td colspan="2" align="center"><a href="/Emp/AddEmp">Add a new Employee</a></td>
  </tr>
</table>

```

Flow of Control => Call `AddEmp` => `Get Action` method of `EmpController` which will launch `AddEmp` => `View` => on click of "Save" button it will invoke `AddEmp` => `Post Action` method and launches `DisplayEmp1` => `View`.

Execute "AddEmp" Action method of "EmpController" as following:

<http://localhost:port/Emp/AddEmp>

Note: in this example, `Action` method "Parameter Names" are exactly matching with "Control Names" of the `View`, so the values entered in the `Controls` will directly come into those `Parameters`. But the drawback in this approach is when there are multiple `Controls` on the `View`, there should also be multiple `Parameters` defined to the `Action` method to read all those `Control Values`, so the `Parameter List` will be `lengthy` and code gets `complex`, to overcome this problem of reading each `Control Value` thru an individual `Parameter` we can use "FormCollection" class as `Action` method parameter, and read values of each `Control` present on the `Form`.

To test this, re-write the "AddEmp" Post Action method as below:

```

[HttpPost]
public ViewResult AddEmp(FormCollection fc)
{
  ViewData["Id"] = fc["id"];
  ViewData["Name"] = fc["name"];
  ViewData["Job"] = fc["job"];
  ViewData["Salary"] = fc["salary"];
  return View("DisplayEmp1");
}

```

Execute "AddEmp" Action method of "EmpController" as following:

<https://localhost:port/Emp/AddEmp>

Note: without reading each controls value thru "FormCollection" we can use "Model Binding" and capture all the values at a time in the `View` and pass them to another `View` for displaying, which provides type safety and intellisense support.

To test this, add a new class under the Models folder with the name "Employee" and write the below code in it:

```

public class Employee
{
  public int? Id { get; set; }
}

```

```

public string Name { get; set; }
public string Job { get; set; }
public double? Salary { get; set; }
}

```

Now go to “EmpController” class and re-write the “AddEmp – Post Action” method as following by importing “MVCUDesigning.Models” namespace.

```

[HttpPost]
public ViewResult AddEmp(Employee Emp)
{
    return View("DisplayEmp2", Emp);
}

```

Add another view with the name “DisplayEmp2” and write the below code in it:

```

@*Write the below statement at top of the View *@
@model MVCUDesigning.Models.Employee

```

```

@*Write this code under the <div> tag.*@
<table border="1" align="center">
    <caption>Employee Details</caption>
    <tr><td>Employee Id:</td><td>@Model.Id</td></tr>
    <tr><td>Employee Name:</td><td>@Model.Name</td></tr>
    <tr><td>Employee Job:</td><td>@Model.Job</td></tr>
    <tr><td>Employee Salary:</td><td>@Model.Salary</td></tr>
    <tr><td colspan="2" align="center"><a href="/Emp/AddEmp">Add a new Employee</a></td></tr>
</table>

```

Run “AddEmp” Action method of “EmpController” with the following URL:

<https://localhost:port/Emp/AddEmp>

Flow of Control => Call **AddEmp** => **Get Action** method of **EmpController** which will launch **AddEmp** => **View** => on click of “Save” button it will invoke **AddEmp** => **Post Action** method and launches **DisplayEmp2** => **View**.

Designing a View (UI) by using HTML Helpers: we are all aware that in **Web Applications** we design **User Interfaces** with the help of **HTML Controls**, and to simplify **UI** designing process, **MVC** provides us “**HTML Helper Methods**” and these methods will **generate or render** required “**HTML Code**” i.e., the return type of these methods is a **string** and that produces required **HTML**. These helper methods are equivalent to “**ASP.NET Server Controls**” that we used in “**ASP.NET Web Forms**”.

Working with Html Helper Methods:

Syntax:	<code>@Html.TextBox(string name)</code>	=> string (Return Type)
Example:	<code>@Html.TextBox("Id")</code>	=> <input type="text" id="Id" name="Id" value="" />

In the above case “`@Html`” is an instance of the class “`System.Web.Mvc.HtmlHelper`” and this class will generate the required **HTML** and returns it in the form of a **string** when we call **methods** of the class, and these methods are called as “**HTML Helper Methods**” and all of them are “**Extension Methods**”.

Methods of the “HtmlHelper” class:

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

Creating a “Form” using Html Helper Methods:

```
BeginForm()  
BeginForm(string actionname, string controllername)  
BeginForm(string actionname, string controllername, object routevalues)  
BeginForm(string actionname, string controllername, FormMethod method)  
BeginForm(string actionname, string controllername, object routevalues, FormMethod method)  
  
@{  
    Html.BeginForm();  
    -Place all controls here  
    Html.EndForm();  
}  
  
Or  
@using(Html.BeginForm())  
{  
    -Place all controls here  
}
```

Creating a Label:

```
@Html.Label(string value)  
@Html.Label(string name, string value)  
@Html.Label(string name, string value, object HtmlAttributes)
```

Creating a TextBox:

```
@Html.TextBox(string name)  
@Html.TextBox(string name, object default)  
@Html.TextBox(string name, object default, object HtmlAttributes)
```

Creating a Password:

```
@Html.Password(string name)
```

```
@Html.Password(string name, string default)
@Html.Password(string name, string default, object HtmlAttributes)
```

To work with “**Html Helpers**” add a new **Controller** under “**MVCUIDesigning**” project naming it as “**AccountController**”, delete the existing code in the **class** and write the below code over there:

```
public ViewResult Login()
{
    return View();
}
public ViewResult Validate()
{
    string Name = Request["txtName"];
    string Password = Request["txtPwd"];
    if (Name == "Admin" && Password == "Admin@123")
    {
        Session["Name"] = Name;
        return View("Success");
    }
    else
    {
        ViewBag.Name = Name;
        return View("Failure");
    }
}
```

Add 3 Views under “**Account**” folder which is present under “**Views**” folder naming them as “**Login.cshtml**”, “**Success.cshtml**” and “**Failure.cshtml**” and write code under them.

Note: if we want to use **Bootstrap** for styling of **HTML** in **Views**, we need to install **Bootstrap** in our projects, and this can be done in 2 different ways i.e., either by using “**NuGet Package Manager**” or “**Library Manager**”.

What is NuGet?

Ans: NuGet can be used to find and install packages, i.e., software libraries, assemblies, and any other things that you want to use in your project. NuGet is not a tool that is specific to “**ASP.NET MVC**” projects. This is a tool that you can use inside of “**Visual Studio**” for “**Console Applications**”, “**Windows Applications**”, “**WPF Applications**” and any other type of **Application**. NuGet is a **Package Manager**, and is responsible for **downloading**, **installing**, **updating**, and **configuring** software in your project. From the term software we don’t mean end users software like **Microsoft Word** or **Notepad**, etc. but any pieces of software, which you want to use in our project i.e., “**Assembly**” references.

NuGet Package Manager: this is an option which provides an interface to search for the libraries that can be used in our project, which also helps in installing those libraries in our projects and consume. To use this, go to **Tools Menu** => Select the Menu Item, “**NuGet Package Manager**” and under that select the Sub Menu Item, “**Manage NuGet Packages for Solution**” and this will launch a new window, choose “**Browse**” option on the top and search for “**Bootstrap**”, which displays various “**Bootstrap**” libraries that are available, select “**bootstrap**”, “**by The Bootstrap Authors, Twitter Inc.,**” and then in the RHS select the “**Checkbox**” that displays our “**Project Name**” and then the

“Install” button is **enabled**, click on it, which will install the required “Bootstrap” files into our application by creating a set of folders under the project with the name “Content” and “Scripts”.

Library Manager: this option was newly added from “Visual Studio 2017” for managing all the client-side libraries and to use this open **Solution Explorer**, right-click on the **Project** and choose **Add => “Client-Side Library”** which will launch “**Add Client-Side Library**” dialog and in that fill the below details:

- Provider: cdnjs (default)
- Library: twitter-bootstrap@5.3.2
- Select Include all library files radio button.
- Target Location: Content/Bootstrap

Click “Install” button and this will install “Bootstrap” in our project under the specified folder, and this action will add a file in the project with a name “libman.json” and under this file the details of all the libraries installed will be present as following:

```
{  
  "version": "1.0",  
  "defaultProvider": "cdnjs",  
  "libraries": [  
    {  
      "library": "twitter-bootstrap@5.3.2",  
      "destination": "Content/Bootstrap"  
    }  
  ]  
}
```

Note: same as the above we can also install any other libraries like **jQuery**, **Ajax**, etc. into our projects by using the above 2 tools.

To use bootstrap in “Login.cshtml”, drag and drop “bootstrap.min.css” file from “Content” folder in to the “<head>” section and then write the below code under the “<div>” tag:

```
@using (Html.BeginForm("Validate", "Account"))  
{  
  @Html.Label("User Name:", new { @class = "text-info" });  
  @Html.TextBox("txtName", "", new { placeholder = "Enter Name" });  
  @Html.Label("Password:", new { @class = "text-info" });  
  @Html.Password("txtPwd", "", new { placeholder = "Enter Password" });  
  <input type="submit" value="Login" class="btn btn-success" />  
  <input type="reset" value="Reset" class="btn btn-success" />  
}
```

Success.cshtml: write the below code under its “<div>” tag.

```
<h4>Hello @Session["User"], welcome to the site.</h4>
```

Failure.cshtml: write the below code under its “<div>” tag.

```
<h4>Hello @ViewData["User"], your given credentials are invalid.</h4>
```

Execute Login Action method of “HomeController” using the below URL:

<https://localhost:port/Account/Login>

This will launch “Login.cshtml” View, to enter user credentials and when we fill the credentials and click on “Login” button it will launch “Success.cshtml” View, if the **credentials** are **valid** or else “Failure.cshtml” View, if the **credentials** are **invalid**.

Designing Views using “Html Helpers” and reading Form values using Models:

Step 1: Add a new class in **Models** folder of “**MVCUIDesigning**” project, naming it as “**Student**” and write the below code in it:

```
public class Student
{
    public int Sid { get; set; }
    public string Name { get; set; }
    public int Class { get; set; }
    public string Gender { get; set; }
    public double Fees { get; set; }
    public string Address { get; set; }
}
```

Step 2: Add a new **Controller** in the **Controllers** folder naming it as “**StudentController**”, delete all the existing content of the class and write the below code over there by importing “**MVCUIDesigning.Models**” namespace for working with “**Student**” class:

```
[HttpGet]
public ViewResult AddStudent()
{
    return View();
}

[HttpPost]
public ViewResult AddStudent(Student student)
{
    return View("DisplayStudent", student);
}
```

Step 3: Add a View for “**AddStudent**” action method and write the below code in it:

```
@*Write this code under the <div> tag.*@
@using (Html.BeginForm())
{
    @Html.Label("Student Id: ") <br />
    @Html.TextBox("Sid") <br />
    @Html.Label("Name: ") <br />
```

```

@Html.TextBox("Name") <br />
@Html.Label("Class: ")
List<int> li = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
SelectList items = new SelectList(li);
@Html.DropDownList("Class", items, "-Select Class-") <br />
@Html.Label("Gender: ")
@Html.RadioButton("Gender", "Male") @:Male
@Html.RadioButton("Gender", "Female") @:Female <br />
@Html.Label("Fees: ") <br />
@Html.TextBox("Fees") <br />
@Html.Label("Address: ") <br />
@Html.TextArea("Address", "", 3, 21, new {})
<br /><br />
<input type="submit" value="Save" />
<input type="reset" value="Reset" />
}

```

Step 4: Add a new View with the name “DisplayStudent” under “Student” folder of “Views” folder and write the below code in it:

```

@*Write the below statement at top of the View *@
@model MVCUIDesigning.Models.Student
@*Write this code under the <div> tag*@
<h4>Student Details</h4>
<ul>
    <li>Student Id: @Model.Sid</li>
    <li>Student Name: @Model.Name</li>
    <li>Student Class: @Model.Class</li>
    <li>Student Gender: @Model.Gender</li>
    <li>Student Fees: @Model.Fees</li>
    <li>Student Address: @Model.Address</li>
</ul>

```

Execute “AddStudent” Action method of “StudentController” using the below URL:

<https://localhost:port/Student/AddStudent>

This will launch “AddStudent” view to enter details for a new “Student” and when we fill the Form and click on the “Save” button, will launch “DisplayStudent.cshtml” View to display the details which the user has entered in the Form.

Strongly Typed Html Helpers: In the above program when we are reading the form values by using **Model Object** in “Post” action method we have a problem, i.e., **Control Name** should exactly be same as the **Property Name** of Model class. For example, if the **Property Name** is “Sid” in **Model Class** then **Control Name** also should be “Sid” only and if we give this as wrong, we don’t get any **compile time or run-time** errors, but that controls value will not be read into the **Model’s Property**, and also **intellisense** will not help us, to identify the **Model Property Name**. To overcome this problem, in “MVC 2.0” we are provided with “**Strongly Typed Html Helper**” and these **Helpers** purely depends on

Model Classes, so while designing the UI's we don't require to give any name to **Control's**. **Strongly Typed Html Helpers** are also same as **Html Helpers** we have used till now, but the only difference is, here every **Helper** method is suffixed with "**For**", for example:

Html Helpers	Strongly Typed Html Helpers
Label	LabelFor
Hidden	HiddenFor
ListBox	ListBoxFor
TextBox	TextBoxFor
TextArea	TextAreaFor
Password	PasswordFor
CheckBox	CheckBoxFor
RadioButton	RadioButtonFor
DropDownList	DropDownListFor
Editor	EditorFor
Display	DisplayFor
DisplayName	DisplayNameFor

To work with Strongly Typed Html Helpers add 2 new action methods in the "StudentController" as following:

```
[HttpGet]
public ViewResult AddStudentST()
{
    return View();
}

[HttpPost]
public ViewResult AddStudentST(Student student)
{
    return View("DisplayStudentST", student);
}
```

Add a View to "AddStudentST" Action method and write the below code in its "<div>" tag:

```
@*Write the below statement at top of the View *@
@model MVCUDesigning.Models.Student

@*Write this code under the <div> tag:*@
@using (Html.BeginForm())
{
    @Html.LabelFor(S => S.Sid); <br />
    @Html.TextBoxFor(S => S.Sid); <br />
    @Html.LabelFor(S => S.Name); <br />
    @Html.TextBoxFor(S => S.Name); <br />
    @Html.LabelFor(S => S.Class);
    List<int> li = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    SelectList items = new SelectList(li);
    @Html.DropDownListFor(S => S.Class, items, "-Select Class-"); <br />
```

```

@Html.LabelFor(S => S.Gender);
@Html.RadioButtonFor(S => S.Gender, "Male")@:Male
@Html.RadioButtonFor(S => S.Gender, "Female")@:Female
@Html.RadioButtonFor(S => S.Gender, "Transgender")@:Transgender <br />
@Html.LabelFor(S => S.Fees); <br />
@Html.TextBoxFor(S => S.Fees); <br />
@Html.LabelFor(S => S.Address); <br />
@Html.TextAreaFor(S => S.Address, 3, 21, new { });
<br /><br />
<input type="submit" value="Save" />
<input type="reset" value="Reset" />
}

```

Add a new View with the name “**DisplayStudentST**” under “**Student**” folder of “**Views**” folder and write the below code in it:

```

@*Write the below statement at top of the View *@
@model MVCUIDesigning.Models.Student

@*Write this code under the <div> tag.*@
<h4>Student Details</h4>
<ul>
<li>@Html.DisplayNameFor(S => S.Sid): @Html.DisplayFor(S => S.Sid)</li>
<li>@Html.DisplayNameFor(S => S.Name): @Html.DisplayFor(S => S.Name)</li>
<li>@Html.DisplayNameFor(S => S.Class): @Html.DisplayFor(S => S.Class)</li>
<li>@Html.DisplayNameFor(S => S.Gender): @Html.DisplayFor(S => S.Gender)</li>
<li>@Html.DisplayNameFor(S => S.Fees): @Html.DisplayFor(S => S.Fees)</li>
<li>@Html.DisplayNameFor(S => S.Address): @Html.DisplayFor(S => S.Address)</li>
</ul>

```

Execute “AddStudentST” Action method of “StudentController” using the below URL:

<https://localhost:port/Student/AddStudentST>

This will launch “**AddStudentST**” view to enter details for a new “**Student**”, when we fill the **Form** and click on “**Save**” button, it will launch “**DisplayStudentST.cshtml**” view to display the details which the user has entered in the **Form**.

Note: In “**DisplayStudentST.cshtml**” we have used **DisplayNameFor** method which will get the **Model Property-Name** over there and **DisplayFor** method will get the **Model Property-Value** over there.

Partial Views

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., “**MVCUDesigning**” 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 the image as “**Header**”.

Now right click on the Shared folder select “**Add**” => “**View**”, name it as “**_Header**” and check the “**Create as a Partial View**” Checkbox which will disable all the other **Checkbox’s** on the window, and then Click on “**Add**” button. This action will create a blank “**.cshtml**” file, write the below code in it:

```
<div style="color:darksalmon;text-align:center;height:150px;background-color:azure;border-style:dotted;
border-color:slateblue;border-width:thick; box-shadow: 0px 0px 50px inset">

<marquee style="font-size:x-large;color:crimson;" behavior="alternate">
    .NET, Java, Python, Hadoop, Data Science, DevOps, etc.
</marquee>
</div>
```

Consuming a Partial View: after creating the **Partial View**, we can consume it in any of our **Views** by calling the below 3 methods of “**HtmlHelper**” class:

- `@Html.Partial(string PartialViewName) => MvcHtmlString`
- `@{ Html.RenderPartial(string PartialViewName); } => void`
- `@{ Html.RenderAction(string ActionMethodName); } => void`

The first option of calling the method “**Partial**” returns an “**MVCHtmlString**” and we can call this method in any of our **View Pages**. Because this method returns an **MVCHtmlString** back there is a chance to **manipulate** the content of the **Partial View** in the future within our **View Pages**.

The second option of calling the method “**RenderPartial**” can also be done in our **View Pages**, and this is a **preferred** option because its return type is **void** and there is no chance to **manipulate** the **Partial View** i.e., it can be used **as is only**.

Third option is also **non-modifiable**, but to use this we again need to define a new **Action** method under any **Controller** class because in this case we call the **Action** method in our **View Page** but not **Partial View**.

Html.Partial: to test this, open “**AddStudent.cshtml**” View of “**StudentController**” and write the below code in its “**<body>**” section just above the “**<div>**” tag:

```
@Html.Partial("_Header")
```

Note: as said above we can manipulate the content of the **Partial View** in this case and to test that, go to “**DisplayStudent.cshtml**” view of “**StudentController**” and write the below code in its “**<body>**” section just above the “**<div>**” tag:

```

@{
    String strHeader = Html.Partial("_Header").ToString();
    strHeader = strHeader.Replace("150", "200");
    strHeader = strHeader.Replace("azure", "pink");
    MvcHtmlString htmlString = new MvcHtmlString(strHeader);
    @htmlString
}

```

In the above case “width” of “<div>” is increased from 150 pixels to 200 pixels and the “background-color” of “<div>” is changed from azure to pink, and in the same way we can manipulate anything with-in the Partial View.

Html.RenderPartial: to test this, open “AddStudentST.cshtml” View of “StudentController” and write the below code in its “<body>” section just above the “<div>” tag:

```
@{ Html.RenderPartial("_Header"); }
```

Html.RenderAction: in this case we can’t directly call the “PartialView”, but what we need to call is Action methods which will in-turn call the “PartialView”. To test this, go to “StudentController” and define a new Action method in it as below:

```

public PartialViewResult Header()
{
    return PartialView("_Header");
}

```

Now go to “DisplayStudentST.cshtml” view of “StudentController” and write the below code in the “<body>” section just above the “<div>” tag:

```
@{ Html.RenderAction("Header"); }
```

Creating a Partial View that loads data from Model:

Step 1: add a new class in **Models** folder of the project naming it as **Customer** and write the below code in it:

```

public class Customer
{
    public int Custid { get; set; }
    public string Name { get; set; }
    public string Account { get; set; }
    public double Balance { get; set; }
    public string City { get; set; }
    public bool Status { get; set; }
    public string Photo { get; set; }
}

```

Step 2: under the **Images** folder of the project add **6 images** corresponding to **6 Customers** naming them as “**Image1**”, “**Image2**”, “**Image3**”, “**Image4**”, “**Image5**” and “**Image6**”.

Step 3: add a new **Controller** under the **Controllers** folder with the name as “**CustomerController**”, delete existing code in the class and write the below code over there, by importing the namespace “**MVCUDesigning.Models**”:

```

public ViewResult DisplayCustomers()
{
    Customer c1 = new Customer { Custid = 1001, Name = "Peter", Account = "Savings", Balance = 50000.00,
        City = "Delhi", Status = true, Photo = "/Images/Image1.jpg" };
    Customer c2 = new Customer { Custid = 1002, Name = "Kevin", Account = "Current", Balance = 50000.00,
        City = "Kolkata", Status = true, Photo = "/Images/Image2.jpg" };
    Customer c3 = new Customer { Custid = 1003, Name = "Sandra", Account = "Demat", Balance = 50000.00,
        City = "Mumbai", Status = true, Photo = "/Images/Image3.jpg" };
    Customer c4 = new Customer { Custid = 1004, Name = "Williams", Account = "Savings", Balance = 50000.00,
        City = "Chennai", Status = true, Photo = "/Images/Image4.jpg" };
    Customer c5 = new Customer { Custid = 1005, Name = "John", Account = "Demat", Balance = 50000.00,
        City = "Bengaluru", Status = true, Photo = "/Images/Image5.jpg" };
    Customer c6 = new Customer { Custid = 1006, Name = "Bill", Account = "Current", Balance = 50000.00,
        City = "Hyderabad", Status = true, Photo = "/Images/Image6.jpg" };

    List<Customer> customers = new List<Customer> { c1, c2, c3, c4, c5, c6 };

    return View(customers);
}

```

In the above case **Display** action method is creating instance of **6 Customers**, adding them into a **List** and then sending that **List** to its corresponding **View** to display. To display each **Customer's** data in a table format lets create a **Partial View** first.

Step 4: add a **Partial View** in the **Shared** folder of the **Views** folder naming it as "**_CustomerView**", select the "**Create as a Partial View**" CheckBox, click **Add** button and write the below code in it:

```

@model MVCUIDesigning.Models.Customer


|                                                                         |                                          |
|-------------------------------------------------------------------------|------------------------------------------|
|  > | Custid: @Model.Custid                    |
|                                                                         | Name: @Model.Name                        |
|                                                                         | Account: @Model.Account                  |
|                                                                         | Balance: @Model.Balance                  |
|                                                                         | City: @Model.City                        |
|                                                                         | Status: @Html.CheckBoxFor(C => C.Status) |


```

Step 5: add a View to "**DisplayCustomers**" action method of "**CustomerController**" and write the below code in it.

```

@*Write the below statement at top of the View *@
@model List<MVCUIDesigning.Models.Customer> Or @model IEnumerable<MVCUIDesigning.Models.Customer>

```

```

@*Write this code under the <div> tag: *@
<h2>Customer Details</h2>
@foreach (MVCUIDesigning.Models.Customer customer in Model) or @foreach (var customer in Model)
{
    Html.RenderPartial("_CustomerView", customer);
}

```

When we send a request to “**DisplayCustomers**” action method of “**CustomerController**”, the method creates a List of **Customers** and transfers it to “**DisplayCustomers.cshtml**” View, and this **View** will in turn call the “**Partial View**” i.e., “**_CustomerView**” for displaying each Customer data in a table format.

Note: **IEnumerable (Interface)** is a parent of all the **Collection** classes, so in place of the **List** we can also use **IEnumerable**, so that going forward this **IEnumerable** can hold the data that is coming from any **Collection** object.

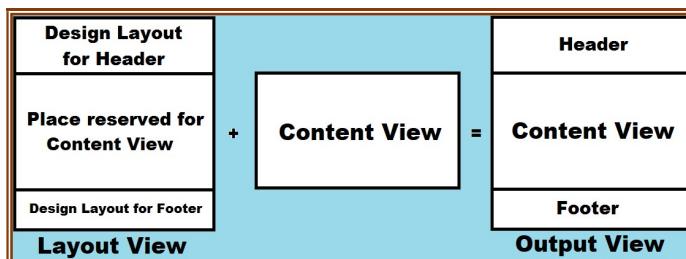
Layout Views

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 Views** so these **Content Pages** or **Content Views** will merge with the **Layout View** to display the final output.

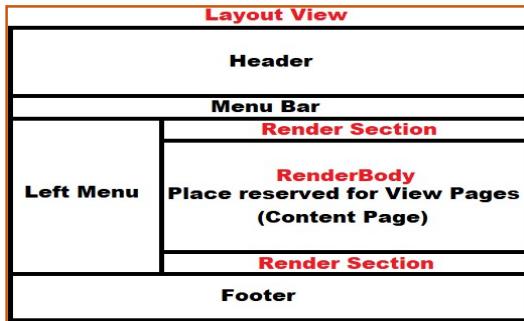


To include a **Content View** into a particular location of a **Layout View** we need to call a method i.e., “**RenderBody**” so that the **Content View** will come and sit in that location. A **Layout View** can contain 1 and only 1 “**RenderBody**” method call.

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.



Designing a Layout View:

Step 1: Create a new “ASP.NET Web Application” project naming it as “MVCLayoutViews” choose “Empty Project Template”, check “MVC” Checkbox and click on the **Create** button to create the project.

Step 2: Install Bootstrap into the project either by using “NuGet Package Manager” or “Library Manager”, which installs all the required files by creating required folders under the project.

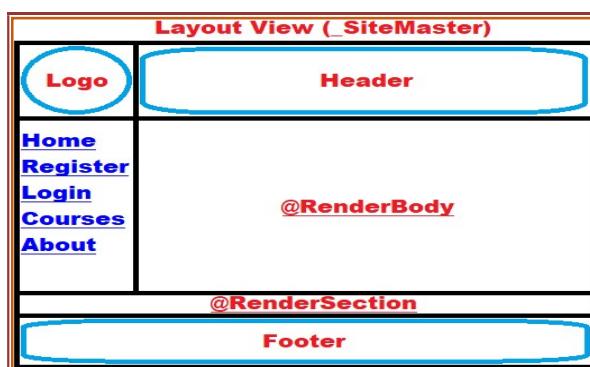
Step 3: Add a new folder under **Views** folder naming it as “Shared” because “Partial Views” and “Layout Views” are generally stored in **Shared** folder only and the name of these 2 **Views** will be prefixed with “_ (underscore)” character.

Step 4: add a **View** into the Shared folder naming it as “**_SiteMaster.cshtml**”, uncheck the CheckBox “**Use a layout page**” and click on the **Add** button. This action will add a new **View** and if we observe the code in this **View**, we find a statement on the top of the file as following to specify a Layout for this **View**:

```

@{
    Layout = null;
}
  
```

- Right now, what we are designing is a “**Layout View**” only and we don’t want to apply another **layout** to the current **View**, so delete it.
- Now under the **<title>** section of the page we find “**_SiteMaster**”, delete it and write the following there: “**@ ViewBag.Title – NIT**” so that we can send any value to this “**Title**” from the “**Content Views**”.
- Now drag and drop “**bootstrap.min.css**” file from the content folder into the “**<head>**” section.
- Add a new folder under the **Project** with the name **Images** and add 3 **Images** into that folder: “**Header.png**”, “**Footer.png**” and “**Logo.jpg**”.
- Now let’s design the **Layout View** as below:



Write the below code under the “<div>” tag to create the above “Layout View”:

```
<table style="width:100%">
<tr>
<td align="center" style="background-color:aquamarine">
    <img src "~/Images/Logo.jpg" style="height:100px; width:100px" class="rounded-circle" />
</td>
<td>
    <img src "~/Images/Header.png" style="height:100px; width:100%" class="rounded" />
</td>
</tr>
<tr>
<td style="vertical-align:top; background-color:burlywood">
    <nav class="navbar-dark bg-primary">
        <font size="4">
            @Html.ActionLink("Home Page", "Home", "NareshIt", new { @class = "navbar-brand" })<br />
            @Html.ActionLink("Registration Page", "Register", "NareshIt", new { @class = "navbar-brand" })<br />
            @Html.ActionLink("Login Page", "Login", "NareshIt", new { @class = "navbar-brand" })<br />
            @Html.ActionLink("Courses Offered", "Courses", "NareshIt", new { @class = "navbar-brand" })<br />
            @Html.ActionLink("About Us", "About", "NareshIt", new { @class = "navbar-brand" })<br />
        </font>
    </nav>
</td>
<td>
    @RenderBody()
</td>
</tr>
<tr>
<td colspan="2" style="background-color:coral; text-align:center">
    @RenderSection("S1", false)
</td>
</tr>
<tr>
<td colspan="2" style="width:100%">
    <img src "~/Images/Footer.png" style="width:100%; height:50px" class="rounded" />
</td>
</tr>
</table>
```

How is a Layout applied to a View (Content View)?

Ans: if we want to apply any **Layout** to a **View**, we can do that in 2 different ways.

1. Page Level Binding: we can do this by writing the below code on the **top** of every **View Page**:

```
@{
    Layout = "~/Views/Shared/_SiteMaster.cshtml";
}
```

Note: while adding a new **View** we can select the “**Layout View**” for the **View** by selecting the **CheckBox**, with the caption “**Use a layout page**” and when we select the **CheckBox**, a **TextBox** and **Button** with the caption “**...** will enable, and by clicking on that **Button** we can select our “**Layout View**” so that it will write the above code automatically. The **drawback** in this approach is we need to use this statement on every “**Content Page**” where we want to apply the **Layout**.

2. Application-Level Binding: we do this by writing code in “**_ViewStart.cshtml**” file because this **View** will execute before execution of any other **View**, and this is called implicitly by **MVC Framework**. This file should be in the root folder of “**Views**” i.e., directly under the “**Views**” folder. Let’s add this file in the **Views** folder and to do that right click on **Views** folder and select the option => **Add => View**, name the **View** as “**_ViewStart.cshtml**”, choose “**Empty (Without model)**” template and make sure all the **Checkboxes** are **un-checked** and click on the **Add Button**. Delete all the content present in the file and write the below code over there:

```
@{  
    Layout = "~/Views/Shared/_SiteMaster.cshtml";  
}
```

Note: if the “**_ViewStart.cshtml**” file is not added by us and when adding a new **View** if we check the **CheckBox** “**Use a layout page**”, but did not select any “**Layout View**” then “**MVC Scaffolding**” will add a “**Layout View**” in to the project with the name “**_Layout.cshtml**” and also it adds “**_ViewStart.cshtml**” file under the **Views** folder with the below code in it:

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

Whereas when we add “**_ViewStart.cshtml**” explicitly and specified the layout as “**_SiteMaster.cshtml**”, then when adding a new **View** if we check the **CheckBox** “**Use a layout page**” but did not select a “**Layout View**” then automatically the Layout we specified in “**_ViewStart.cshtml**” will be applied i.e., “**_SiteMaster.cshtml**”.

Note: if the “**_ViewStart.cshtml**” file is present under the **Views** folder, then all the **Views** will apply this layout whereas if it is present under the **Views** folder corresponding to a **Controller** then it applies only to the **Views** in that folder and if we have this file present in both the places then first preference goes to **local**.

Step 5: Add a new **Controller** in **Controllers** folder naming it as “**NareshITController**”, delete the existing **Action** method in it and write the below code over there:

```
public ViewResult Home()  
{  
    return View();  
}  
public ViewResult Register()  
{  
    return View();  
}  
public ViewResult Login()  
{  
    return View();  
}
```

```

public ViewResult Courses()
{
    return View();
}
public ViewResult About()
{
    return View();
}

```

Step 6: Add **Views** to all the Action methods and while adding them check the CheckBox “**Use a layout page**”.

Note: as said above, when we perform application level binding of a “**Layout View**” by entering the details of the “**Layout View**” under “**_ViewStart.cshtml**” file then while adding a **View** if we select the CheckBox “**Use a layout page**” then it will automatically make use of the “**Layout View**” that is specified in “**_ViewStart.cshtml**”, whereas if we have multiple “**Layout Views**”, we need to click on “**...**” Button and select the “**Layout View**” explicitly which will then perform **Page Level Binding**.

Code under “Home.cshtml” file:

```

@{
    ViewBag.Title = "Home Page";
}

<p style="text-align: justify; color: palevioletred; font-family: Arial; text-indent: 50px">We have set the pace with
online learning. Learn what you want, when you want, and practice with the instructor-led training sessions, on-
demand video tutorials which you can watch and listen.</p>
<ol style="color:blueviolet; background-color: darkseagreen">
    <li>150+ Online Courses</li>
    <li>UNLIMITED ACCESS</li>
    <li>EXPERT TRAINERS</li>
    <li>ON-THE-GO-LEARNING</li>
    <li>PASSIONATE TEAM</li>
    <li>TRAINING INSTITUTE OF CHOICE</li>
</ol>
<p style="text-align: justify; color: palevioletred; font-family: Arial; text-indent: 50px">Naresh I Technologies is
renowned around the world for classroom training where every aspirant is encouraged to attend technical sessions
discussing every facet of the subject with excruciating detail on varied technologies. Every aspirant in the training
program will be provided with the hands-on experience through several Lab assignments and case studies there by
making them more employable.</p>
@section S1 {
    <font size="3" color="blue">Excellence in IT Training since 2003.</font>
}

```

Code under “Register.cshtml” file:

```

@{
    ViewBag.Title = "Registration Page";
}

<h3 style="text-align:center;text-decoration:underline">Registration Page</h3>
<table align="center" class="table-bordered">
```

```

<tr>
    <td>Name:</td><td><input type="text" id="txtName" /></td>
</tr>
<tr>
    <td>User Id:</td><td><input type="text" id="txtUid" /></td>
</tr>
<tr>
    <td>Password:</td><td><input type="password" id="txtPwd" /></td>
</tr>
<tr>
    <td>Confirm Password:</td><td><input type="password" id="txtCPwd" /></td>
</tr>
<tr>
    <td>Mobile:</td><td><input type="tel" id="txtMobile" /></td>
</tr>
<tr>
    <td>Email Id:</td><td><input type="email" id="txtEmail" /></td>
</tr>
<tr>
    <td colspan="2" align="center">
        <input type="submit" id="btnRegister" value="Register" />
        <input type="reset" id="btnReset" value="Reset" />
    </td>
</tr>
</table>
@section S1 {
    <font size="3" color="blue">Join us for best IT Training.</font>
}

```

Code under “Login.cshtml” file:

```

@{
    ViewBag.Title = "Login Page";
}

<h3 style="text-align:center;text-decoration:underline">Login Page</h3>
<table class="table-bordered" align="center">
    <tr><td>User Id:</td><td><input type="text" id="txtUid" /></td></tr>
    <tr><td>Password:</td><td><input type="password" id="txtPwd" /></td></tr>
    <tr>
        <td colspan="2" align="center">
            <input type="submit" id="btnLogin" value="Login" />
            <input type="reset" id="btnReset" value="Reset" />
        </td>
    </tr>
</table>
@section S1 {
    <font size="3" color="blue">Trained over 10,00,000 IT Professionals.</font>
}

```

Code under “Courses.cshtml” file:

```
@{  
    ViewBag.Title = "Courses Page";  
}  


## Courses Offered:



- .NET
- Java
- Python
- DevOps
- Selenium
- Cloud Technologies
- Data Science
- Automation
- Software Testing
- Mobile Technologies
- Digital Marketing
- Spring Boot & Microservices



@section S1 {  
    Experienced faculties with excellent training skills.  
}


```

Code under “About.cshtml” file:

```
@{  
    ViewBag.Title = "About Page";  
}  


NareshIT (Naresh I Technologies) is a leading Software Training Institute and provides Job Guarantee Program through Nacre in Hyderabad, Chennai, Bangalore, Vijayawada and across the world with Online Training services. Managed and Lead by IT Professionals with more than a decade experience in leading MNC companies. We are most popular for our training approach that enables students to gain real-time exposure on cutting-edge technologies. Naresh I Technologies started satisfying the students who are at remote locations with Online Training program, with an intention to provide services for all those aspirants who want to be part of the software industry. The aspirants who due to various reasons cannot travel physically to our location can just keep themselves registered for this program along with their needs with just a single phone call or a mail provided in our contact's page. The Online Training program is conducted over the internet using the latest state of the art software tools and via the video and desktop sharing facility. The training program is conducted by well experienced training faculty who are mentored in conducting such sessions as per the convenience of the participant. The Online Training program is provided with required material for reference and on hands exercises and Lab sessions for practical exposure. The students are monitored for any clarifications and doubts by the faculty all through the course either by phone or mail as per the convenience of the participants. Our sole maxim in introducing Online Technical Training Sessions is to cater to the training requirements of the aspirants who due to various reasons cannot travel physically to our location. In the online training program, the training sessions are conducted over the internet using the latest state of the art software tools and the hands-on experience also will be provided at the convenience of the participant.



</p>


```

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
HttpStatusActionResult	HttpUnauthorizedResult
RedirectToRoute	RedirectToRouteResult
RedirectToAction	RedirectToRouteResult

- ViewResult: sends a view as response.
- PartialViewResult: sends a partial view as response.
- 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 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.
- HttpStatusActionResult: provides a way to return an action result with a specific **HTTP Status Code** and Description.

In the previous programs we have already used “ViewResult”, “PartialViewResult” and “RedirectToRouteResult” and to test the other Action Results lets create a new “ASP.NET Web Application” project naming it as “ActionResultsInMVC”, choose “MVC Project Template”, un-check all the other Checkboxes and click on Create button to create the project.

Note: if we open a project by choosing “MVC Project Template” then, we will find a “Content” folder with “Bootstrap” installed and we will also find a “Scripts” folder with “JQuery” installed in it. In the “Controllers” folder we find a Controller with the name “HomeController” with 3 Action methods in it with the name: Index, About and Contact; and these methods will also have associated Views in the “Home” folder of “Views” folder. We will also have “Shared” folder with a “Layout View (_Layout.cshtml)” and “Error View (Error.cshtml)”, apart from that we will also have “_ViewStart.cshtml” in which the Layout is applied.

Now do the following under the Project:

Step 1: Add a new folder in the project, naming it as “Downloads”, and add 1 “.pdf” and 1 “.doc” file into the folder.

Step 2: Add a new class in **Models** folder of the project naming it as “Employee” and write the below code in it:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Job { get; set; }
    public double Salary { get; set; }
    public bool Status { get; set; }
}
```

Step 3: Add a **Controller** naming it as “ResultsController” and define the below “Action Methods” in the class.

```
using System.Text;
using ActionResultsInMVC.Models;
public JsonResult GetEmployees()
{
    Employee e1 = new Employee { Id = 101, Name = "Scott", Job = "CEO", Salary = 25000, Status = true };
    Employee e2 = new Employee { Id = 102, Name = "Smith", Job = "President", Salary = 22000, Status = true };
    Employee e3 = new Employee { Id = 103, Name = "Parker", Job = "Manager", Salary = 18000, Status = true };
    Employee e4 = new Employee { Id = 104, Name = "John", Job = "Salesman", Salary = 10000, Status = true };
    Employee e5 = new Employee { Id = 105, Name = "David", Job = "Clerk", Salary = 5000, Status = true };
    Employee e6 = new Employee { Id = 106, Name = "Maria", Job = "Analyst", Salary = 12000, Status = true };
    List<Employee> Emps = new List<Employee> { e1, e2, e3, e4, e5, e6 };
    return Json(Emps, JsonRequestBehavior.AllowGet);
}
public FileResult DownloadPdf()
{
    return File("~/Downloads/Test.pdf", "application/pdf");
}
public FileResult DownloadWord()
{
    return File("~/Downloads/Test.doc", "application/msword");
}
public RedirectResult OpenFaceBook()
{
    return Redirect("https://facebook.com");
}
public RedirectResult OpenTwitter()
{
    return Redirect("https://twitter.com");
}
public string SendDate1()
```

```

        return "Current Date: " + DateTime.Now;
    }
    public ContentResult SendDate2()
    {
        return Content("Current Date: " + DateTime.Now.ToString());
    }
    public string SayHello1()
    {
        return "नमस्ते आप कैसे हैं";
    }
    public ContentResult SayHello2()
    {
        return Content("नमस्ते आप कैसे हैं", "text/plain", Encoding.Unicode);
    }
    public JavaScriptResult AlertUser()
    {
        return JavaScript("alert('Hello, how are you.')");
    }
    public void ReturnEmpty1()
    {
        string str = ("Hello World").ToUpper();
    }
    public EmptyResult ReturnEmpty2()
    {
        return new EmptyResult();
    }
}

```

Add a “View” to “Index Action Method” without choosing a Layout and write the below code in its “<div>” tag:

```

<h2>MVC Action Results</h2>
<ol>
    <li>@Html.ActionLink("Json Result", "GetEmployees")</li>
    <li>@Html.ActionLink("Download Pdf", "DownloadPdf", null, new { target = "_blank" })</li>
    <li>@Html.ActionLink("Download Word", "DownloadWord")</li>
    <li>@Html.ActionLink("Facebook", "OpenFaceBook", null, new { target = "_blank" })</li>
    <li>@Html.ActionLink("Twitter", "OpenTwitter", null, new { target = "_blank" })</li>
    <li>@Html.ActionLink("Today's date returned as ContentResult", "SendDate1")</li>
    <li>@Html.ActionLink("Today's date returned as String", "SendDate2")</li>
    <li>@Html.ActionLink("नमस्ते बोलो returned as String", "SayHello1")</li>
    <li>@Html.ActionLink("नमस्ते बोलो returned as ContentResult", "SayHello2")</li>
    <li>@Html.ActionLink("Javascript Alert", "AlertUser")</li>
    <li>@Html.ActionLink("Empty Result", "ReturnEmpty1")</li>
    <li>@Html.ActionLink("Void Result", "ReturnEmpty2")</li>
</ol>

```

Run the “Index” action method of “ResultsController” which provides links to execute the other action methods defined in the class.

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 **JS** Files.)

Note: most of the current modern **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, line breaks, 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.

How to perform Minification?

Ans: To perform **Minification** of assets (**CSS** and **JS files**) we need to take the help of some tool and there are various tools available in the industry to do this. In our code we are going to use “**Portable Smalify**” tool which can be download from this location:

<https://www.softpedia.com/get/PORTABLE-SOFTWARE/System/File-management/Portable-Smalify.shtml>

After downloading, unzip the file into a folder and in that folder, we find “**Smalify.exe**”. Run it, which will open a User Interface, drag, and drop the folder in which we have placed the “**.css**” or “**.js**” files on the UI, which we want to **minify** and click on the “**Minify now**” button on the **UI**.

Step 1: Create a new “**ASP.NET Web Application**” project naming it as “**MinificationAndBundling**”; choose “**Empty Project Template**”, check “**MVC**” checkbox and click on the “**Create**” button.

Add a folder under the project with the name “**Content**” and under this folder add 2 “**.css**” files with the name “**Header.css**” & “**Paragraph.css**”, and write the below code in those files:

Header.css:

```
h1 {
    color: red;text-align: center;background-color: palegreen;border-style: double; border-bottom-color: bisque;
    border-top-color: coral; border-left-color: coral; border-right-color: bisque;text-decoration: underline;
}
```

Paragraph.css:

```
p {  
    text-align: justify; text-justify: inter-word;  
    color: green; font-family: courier; font-size: 160%;  
    text-transform: capitalize; text-indent: 50px; line-height: 1.8; border: 2px solid #0066FF;  
    border-top-width: 2px; border-right-width: 2px; border-bottom-width: 2px; border-left-width: 2px;  
    border-top-style: solid; border-right-style: solid; border-bottom-style: solid; border-left-style: solid;  
    border-top-color: #0066FF; border-right-color: #0066FF; border-bottom-color: #0066FF;  
    border-left-color: #0066FF;  
}
```

Step 2: Minification of “.css” files and to do that right click on **Project** in **Solution Explorer**, select “Open Folder in File Explorer” which will display the list of folders and files under project folder in **Windows Explorer**. Run “Smalify.exe”, drag and drop the “Content” folder on to the UI of the “Smalify.exe” and click on “Minify now” button, which will generate the minified files “Header.min.css” and “Paragraph.min.css”. To load these files in our project, select “Content” folder in **Solution Explorer** and choose the option “Show All Files” in the top of **Solution Explorer** which will show the 2 files under **Content** folder, right click on them and select the option “Include in Project”, which will load them into the **project**, and after loading again click on “Show All Files”.

Step 3: Add a new **Controller** with the name “HomeController” and add a **View** to the existing **Index Action Method** without selecting “Use a layout page” Check Box. Now drag and drop “Header.min.css” and “Paragraph.min.css” files under the “<head>” tag of the **View** and write the below code under the “<div>” tag of the **View**:

```
<h1>Naresh I Technologies</h1>  
<p>Classroom Training schedule is a custom-made training program to suit the aspirants from different educational backgrounds, the training programs are conducted by well-trained faculties in the corresponding specializations in a real-time environment. The classroom training is conducted exclusively in the campus of Naresh i Technologies with well equipped and configured architecture to give students exactly an environment of industrial exposure.</p>  
<p>Naresh i Technologies is renowned around the world for classroom training where every aspirant is encouraged to attend technical sessions discussing every facet of the subject with excruciating detail on varied technologies. Every aspirant in the training program will be provided with the hands-on experience through a number of Lab assignments and case studies there by making them more employable.</p>  
<p>Digital marketing is an umbrella for an online system. Digital marketing refers to advertising which is delivered through the digital channel. Few digital marketing is like websites, social media, emails and mobile apps. The product is being marketed through digital technology. This Digital marketing online training course will help how to use the Facebook, Twitter, LinkedIn, etc for online advertisement. In simple term, the digital marketing means the promotion of the products or brands via some electronic media.</p>  
<p>Corporate Training program is tailor-made to suit the working professionals in the industry who are expected to have a shift in their domain or technology as per their career demand. The corporate training courses are customized to meet the project requirements as expected from the corporate trainees.</p>  
<p>The Corporate Training is provided either at the client location or at our Naresh i Technologies campus depending upon the need of the client. In the global markets of competition training and development of the employees to keep up their technical expertise has become the most important composition of any I.T. Organization.</p>  
<p>The corporate employees require regular enhancements and update of the up-coming technologies in accordance with the advancements as demanded by the end clients. This requirement of the corporate sector is the
```

key juncture where Naresh iTechnologies started sharing its expertise in reducing the training over-heads of the corporate sector.</p>

<p>The success of Naresh i Technologies lies in customized course content depending on the project requirements of the client and the experience level of the participants thereby enhancing the productivity of the overall industry.</p>

<h1>Excellence In IT Training</h1>

Now run the “Index View” and watch the output, because the page is using **2 CSS Files (Assets)** there will be total **3 requests** sent from the **browser to server** to load the **assets** as following:

- Index.cshtml
- Header.min.css
- Paragraph.min.css

To watch this hit “F12” in the **Browser** which will open a window on the “RHS” or “Bottom” of the **Browser** called as “**Developer Tools**”, in that window go to “**Network**” tab, and click on the **refresh** button on the **Browser** which will show the no. of requests in the bottom on “**Developer Tools**” window.

Note: To minimize the no. of requests from browser to server, we can take the help of **bundling**, so that both the **CSS** files will be loaded in a single request and the no. of requests can be reduced to **2** from **3** in the current case.

Bundling: this is a new feature introduced in “ASP.NET 4.5”. **Bundling** makes it easy to **combine** or **bundle** multiple files into a single unit, and by using this we can create **CSS**, **Java Script** and other **bundles**. Fewer files mean fewer **HTTP** requests and that can improve **page load performance**.

To perform **Bundling** we need to take the help of “**Web Optimization Framework**” library in our project and to use it we need to install it thru “**NuGet Package Manager**”, and to do that, open “**NuGet Package Manager**”, go to **Browse** and search for “**Web Optimization Framework**” that displays “**Microsoft.AspNet.Web.Optimization**”, select it and **install** it.

Working with Microsoft AspNet Web Optimization Framework: in this library we are provided with a set of classes under “**System.Web.Optimization**” namespace:

- **StyleBundle:** this represents a collection of **Style Sheet** files.
- **ScriptBundle:** this represents a collection of **Script** files.
- **BundleCollection:** this represents a set of “**StyleBundles**” and “**ScriptBundles**”.
- **BundleTable:** this is a holder class for the “**BundleCollection**”.
- **Scripts:** this class is used for **rendering** the **Script Bundles** in our **Web Pages** i.e., “**.cshtml**” files and to do that we need to call a **static** method under the class i.e., **Render** to render the **script** files in **ScriptBundles**.
- **Styles:** this class is used for **rendering** the **Style Bundles** in our **Web Pages** i.e., “**.cshtml**” files and to do that we need to call a **static** method under the class i.e., **Render** to render the **style** files in **StyleBundles**.

Step 4: To perform bundling, add a “**Code File**” under “**App_Start**” folder with the name “**BundleConfig.cs**” and write the below code in the file:

```
using System.Web.Optimization;  
namespace MinificationAndBundling
```

```

{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            //Create a StyleBundle and include all the CSS files into it
            StyleBundle styleBundle = new StyleBundle("~/Content/NITStyleBundle");
            styleBundle.Include("~/Content/Header.min.css", "~/Content/Paragraph.min.css");
            bundles.Add(styleBundle);

            //Create a ScriptBundle and include all the Scripts files into it (Example Code)
            //ScriptBundle scriptBundle = new ScriptBundle("~/Scripts/NITScriptBundle");
            //scriptBundle.Include("~/Scripts/Test1.min.js", "~/Scripts/Test2.min.js");
            //bundles.Add(scriptBundle); //Example Code
        }
    }
}

```

Step 5: Now open the “Global.asax” file and call “RegisterBundles” method of “BundleConfig” class inside of “Application_Start” method by importing “System.Web.Optimization” namespace:

```
BundleConfig.RegisterBundles(BundleTable.Bundles);
```

Step 6: go to “Index.cshtml” file and do the following:

- Import the namespace “System.Web.Optimization” as following on top of the file:

```
@using System.Web.Optimization;
```

- Delete the 2 <link> tag statements in the head section and write the below code over there:

```
@Styles.Render("~/Content/NITStyleBundle")
```

- Now run the “Index.cshtml” file again and watch the output, but now also we will have 3 requests only because bundling is performed only in the “Release or Production” mode but not in “Debug or Development” mode. If we want to test bundling at the time of “Application Development” we can achieve that in 2 different ways:

- ❖ Open “Web.config” file and there under “<system.web>” tag we find “<compilation>” tag with a boolean attribute “debug” and its value will be “true” by default, change it as “false” and run “Index.cshtml” file again. The code in “Web.config” should now look as following:

```
<compilation debug="false" targetFramework="4.8" />
```

- ❖ With-out changing the “debug” attribute value as “false” in “Web.config” file also, we can perform bundling i.e., in “Debug or Development” mode bundling can be performed and to do that, change the “false” value as “true” again in “Web.config” file, go to “BundleConfig.cs” and write the below statement in the end of “RegisterBundles” method and run “Index.cshtml” page again.

```
BundleTable.EnableOptimizations = true;
```

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 organization 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

To work with areas, create a new “**ASP.NET Web Application**” project naming it as “**MVCAreas**”, choose “**Empty Project Template**”, check the “**MVC**” **CheckBox** and click on “**Create**” button to create the project. To add an **Area** under the project, right click on the **Project** and select the option **Add => New Scaffolded Item**, this will open a new window and in that window on the **LHS** expand the “**MVC Node**” and select the option “**Area**” and then on the **RHS** it will display “**MVC 5 Area**”, select it and click on “**Add**” button which will ask for a name to the **Area**, enter the name as “**Patient**” and click on the “**Add**” button which will add “**Areas**” folder under the **Project** and under that “**Areas**” folder it will add another folder with the name “**Patient**” and under this it will add “**Controllers**”, “**Views**” and “**Data (Models)**” folders and also 1 file with name “**PatientAreaRegistration.cs**” and under this we find a class “**PatientAreaRegistration**” inheriting from “**AreaRegistration**” class and under this class we find a method “**RegisterArea**” defining the route to accessing the “**Action Methods**” in this area as following:

```
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Patient_default",
        "Patient/{controller}/{action}/{id}",
```

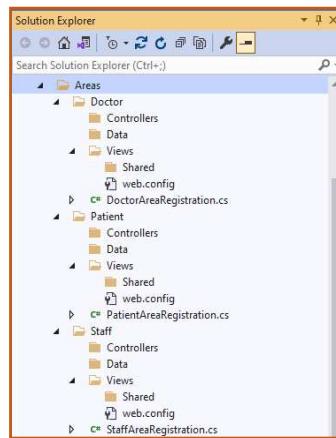
```

    new { action = "Index", id = UrlParameter.Optional }
);
}

```

Adding multiple areas in the Project: to add more “Areas” in the project, now directly right click on the “Areas” in Solution Explorer and select “Add” => “Area” and in the window opened select “MVC 5 Area” and give a name to it, and by following this process add 2 new Areas with the names “Doctor” and “Staff”.

Note: under every Area we have added, we will find “Controllers”, “Views” and “Data” folders as well as “AreaRegistration” class as following:



Add a Controller in each Controllers folder with-in the “Areas” naming them as “HomeController” and add a View to each “Index Action Method” by choosing a layout and write the below code in them by deleting the existing code:

Doctor Home Controller's - Index.cshtml:

```

@{
    ViewBag.Title = "Doctors - Index Page";
}
<h2>This is Home Page of Doctor's Area.</h2>
@Html.ActionLink("Back to site home page", "Index", "Home", new { area = "" }, null)

```

Patient Home Controller's - Index.cshtml:

```

@{
    ViewBag.Title = "Patients - Index Page";
}
<h2>This is Home Page of Patient's Area.</h2>
@Html.ActionLink("Back to site home page", "Index", "Home", new { area = "" }, null)

```

Staff Home Controller's - Index.cshtml:

```

@{
    ViewBag.Title = "Staff - Index Page";
}

```

```
<h2>This is Home Page of Staff Area.</h2>
@Html.ActionLink("Back to site home page", "Index", "Home", new { area = "" }, null)
```

Add a **Controller** under the project's root “**Controllers**” folder naming it as “**HomeController**” only, add a **View** to existing “**Index Action Method**”, choosing a **layout** and write the below code in it by deleting existing code:

```
@{
    ViewBag.Title = "Site - Index";
}
<h2>This is the Home Page of Site.</h2>
@Html.ActionLink("Doctor Home Page", "Index", "Home", new { area = "Doctor" }, null)
<br />
@Html.ActionLink("Patient Home Page", "Index", "Home", new { area = "Patient" }, null)
<br />
@Html.ActionLink("Staff Home Page", "Index", "Home", new { area = "Staff" }, null)
```

Run the project which must launch the above “**Index.cshtml**” View of the root “**Home Controller**” but it will not launch because there are multiple “**Home Controllers**” and “**Index Action Methods**” under the project, so we get an error. To resolve the problem open “**RouteConfig.cs**” file present in “**App_Start**” folder of the project and specify the namespace of our Root “**HomeController**” to it in “**routes.MapRoute**” method, which should now look as below:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    namespaces: new string[] { "MVCAreas.Controllers" }
);
```

This action should now launch the “**Site Home Page**” i.e., “**Index Page**” of root “**Home Controller**” of the Project and in that “**Page**” we will find links to launch “**Doctor’s Home Page**”, “**Patient’s Home Page**” and “**Staff Home Page**”, click on them and test it.

When we enter the below URL in the browser:

<http://localhost:port> => Launches Site Home Page

Same as above when we enter below URL’s they should launch their corresponding Home Pages, for example:

http://localhost:port/Doctor	=>	should launch Doctor home page
http://localhost:port/Patient	=>	should launch Patient home page
http://localhost:port/Staff	=>	should launch Staff home page

But when to try them, we get “**Page Not Found**” error and to resolve this problem go to each “**AreaRegistration.cs**” file that are present in each “**Areas**” folder and there we find the method “**RegisterArea**”, and in that method with-in the last line specify the default **Controller** name i.e., “**Home**”, as following:

Old Code: new { action = "Index", id = UrlParameter.Optional }
New Code: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

Data Annotations

These are nothing but **validations** that we put on 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 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 these 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.

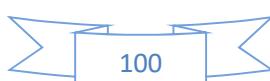
To perform the **Client-Side Validations**, **Data Annotations** uses **JQuery Library**, so our **View Pages** have to use the below **JQuery** libraries:

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

Display Attribute: by using this **attribute** we can specify a **name** that has to be **displayed** by a **Label** on the **View** for the **Model’s Property**.

```
[Display(Name = "First Name")]
public string FirstName { get; set; }
```

DisplayFormat Attribute: by using this **attribute** we can specify how **data fields** are **displayed** and **formatted**, and with the help of this we can do the following:



- Display the text as “[Null]” when a data field is empty.
- Display currency data in locale-specific currency format.
- Display date information in any pre-defined or specified format.

To display numeric data in Currency Format:

```
[DisplayFormat(DataFormatString = "{0:c}")]
public double Salary { get; set; }
```

To display data in Date Format (MM/dd/yyyy):

```
[DisplayFormat(DataFormatString = "{0:d}")]
public DateTime DOB { get; set; }
```

To display data in specified Date Format:

```
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}")]
public DateTime DOB { get; set; }
```

To display data in specified Date & Time format (Time in 24 hour format):

```
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy HH:mm:ss}")]
public DateTime DOB { get; set; }
```

To display data in specified Date & Time format (Time in 12 hour format with AM/PM):

```
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy hh:mm:ss tt}")]
public DateTime DOB { get; set; }
```

To display “Value not specified” when value is null:

```
[DisplayFormat(NullDisplayText = "Name not specified.")]
public string Name { get; set; }
```

Validation Attributes:

Required Attribute: this attribute is used to specify that the value is mandatory and cannot be skipped.

```
[Required(ErrorMessage = "Name field can't be left empty.")]
public string Name { get; set; }
```

DataType Attribute: this attribute is used to specify the data type for the model’s property.

```
[DataType(DataType.Password)]
public string Password { get; set; }
```

Compare Attribute: this attribute is used to compare the value of 1 Model property with other property.

```
[Compare("Password", ErrorMessage = "Confirm password should match with password.")]
public string ConfirmPassword { get; set; }
```

Range Attribute: this attribute is used to specify a range for the Model properties value.

```
[Range(18, 60, ErrorMessage = "Age should be between 18 to 60 years.")]
public int Age { get; set; }
```

RegularExpression Attribute: this attribute is used to compare the Model properties value with a “RegularExpression”.

```
[RegularExpression(@"[6-9]\d{9}")] //Mobile No. should start with 6, 7, 8, and 9 only and can be
having a length of 10 digits (Both Max & Min).
public string Mobile { get; set; }
```

Remote Attribute: this attribute class is defined in “`System.Web.Mvc`” namespace which is used to make “AJAX” call to `Server` and `validate` data without posting the entire form to the server, whenever a server-side validation is preferable in any particular scenario and in this case the result is sent back from server to client in `JSON` format.

To test all the above, create a new “`ASP.NET Web Application`” project naming it as “`MVCDataAnnotations`”, choose “`Empty Project Template`” and select “`MVC`” Checkbox to create the project.

Step 1: Add a `Model` class under the `Models` folder naming it as “`User.cs`” and write the below code in it by importing “`System.ComponentModel.DataAnnotations`” namespace:

```
public class User
{
    [Display(Name = "User Name")]
    [Required(ErrorMessage = "User name field can't be left empty.")]
    [RegularExpression(@"[A-Za-z\s]{3,}", ErrorMessage = "Name can have alphabets & spaces with min size of 3.")]
    public string Name { get; set; }

    [DataType(DataType.Password)]
    [Required(ErrorMessage = "Password field can't be left empty.")]
    [RegularExpression(@"^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$_-])(?=[$+$.]{8,16})", ErrorMessage = "Password
Format: <br />-No spaces.<br />-Minimum 1 numeric.<br />-Minimum 1 upper case & lower case alphabet.
<br />-Minimum 1, any of these Special characters: -, _, @, #, $.<br />-Should be ranging between 8 - 16 chars.")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm Password")]
    [Required(ErrorMessage = "Confirm password field can't be left empty.")]
    [Compare("Password", ErrorMessage = "Confirm password should match with password.")]
    public string ConfirmPassword { get; set; }

    [DataType(DataType.Date)]
    [Display(Name = "Date of Birth")]
    [Required(ErrorMessage = "Date of birth field can't be left empty.")]
    [System.Web.Mvc.Remote("IsValidDate", "User", ErrorMessage = "User should attain 18 years age to register.")]
    public DateTime DOB { get; set; }

    [Display(Name = "Mobile No")]
    [DataType(DataType.PhoneNumber)]
    [Required(ErrorMessage = "Mobile No. field can't be left empty.")]
    [RegularExpression(@"[6-9]\d{9}", ErrorMessage = "Mobile No. should start with 6, 7, 8, and 9 only and can be
having a length of 10 digits (both Max & Min.)")]
    public string Mobile { get; set; }
```

```

[Display(Name = "Email Id")]
[DataType(DataType.EmailAddress)]
public string Email { get; set; }

[Display(Name = "User Address")]
[DataType(DataType.MultilineText)]
public string Address { get; set; }
}

```

Password rules Regular Expression:

(?=.*[0-9]) represents a digit must occur at least once.
 (?=.*[a-z]) represents a lower case alphabet must occur at least once.
 (?=.*[A-Z]) represents an upper case alphabet that must occur at least once.
 (?=.*[@#\$_-]) represents a special character that must occur at least once.
 (=?\S+\\$) white spaces are not allowed in the entire string.
 .{8, 16} represents at least 8 characters and at most 16 characters allowed.

Step 2: Add a **Controller** class in **Controllers** folder of the project naming it as “**UserController**” delete all the content in the class and write the below code in it by importing “**MVCDataAnnotations.Models**” namespace:

```

//Validation code for DOB field which is called using Remote Attribute
public JsonResult IsValidDate(DateTime DOB)
{
    bool Status;
    if (DOB > DateTime.Now.AddYears(-18))
    {
        Status = false;
    }
    else
    {
        Status = true;
    }
    return Json(Status, JsonRequestBehavior.AllowGet);
}
public ViewResult AddUser()
{
    return View();
}
public ViewResult DisplayUser(User user)
{
    return View(user);
}

```

Step 3: Install latest versions of **jQuery**, **jQuery.Validation**, **Microsoft.jQuery.Unobtrusive.Validation**, and **Bootstrap** libraries using “**Nuget Package Manager**”.

Step 4: Add a view to “**AddUser**” action method without choosing any “**Layout**”, and do the following:

[@* Write the below statement at top of the View. *@]

@model MVCDataAnnotations.Models.User

[@* Drag and Drop the following files from Scripts folder into the "<head>" section. *@]

```
<script src="~/Scripts/jquery-3.7.1.min.js"></script>
<script src="~/Scripts/jquery.validate.min.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

[@* Drag and Drop the following file from Content folder into the "<head>" section. *@]

```
<link href="~/Content/bootstrap.min.css" rel="stylesheet" />
```

[@* Write the below code under "<div>" tag. *@]

```
@using (Html.BeginForm("DisplayUser", "User"))
{
    <div>
        @Html.LabelFor(U => U.Name)<br />
        @Html.EditorFor(U => U.Name)
        @Html.ValidationMessageFor(U => U.Name)
    </div>
    <div>
        @Html.LabelFor(U => U.Password)<br />
        @Html.EditorFor(U => U.Password)
        @Html.ValidationMessageFor(U => U.Password)
    </div>
    <div>
        @Html.LabelFor(U => U.ConfirmPassword)<br />
        @Html.EditorFor(U => U.ConfirmPassword)
        @Html.ValidationMessageFor(U => U.ConfirmPassword)
    </div>
    <div>
        @Html.LabelFor(U => U.DOB)<br />
        @Html.EditorFor(U => U.DOB)
        @Html.ValidationMessageFor(U => U.DOB)
    </div>
    <div>
        @Html.LabelFor(U => U.Mobile)<br />
        @Html.EditorFor(U => U.Mobile)
        @Html.ValidationMessageFor(U => U.Mobile)
    </div>
    <div>
        @Html.LabelFor(U => U.Email)<br />
        @Html.EditorFor(U => U.Email)
        @Html.ValidationMessageFor(U => U.Email)
    </div>
    <div>
        @Html.LabelFor(U => U.Address)<br />
    </div>
}
```

```

@Html.EditorFor(U => U.Address)
</div>
<div>
    <input type="submit" value="Save User" class="btn btn-primary" />
    <input type="reset" value="Reset Data" class="btn btn-danger" />
</div>
}

```

Note: Data Annotations will perform Data Validations when we bind Views with Model Classes using the “Strongly Typed Html Helpers” like LabelFor, TextBoxFor, EditorFor, etc.

Now run the View, fill in the details and click on “Save User” button to launch “DisplayUser” View but in this process if we enter any wrong values into the controls or did not enter values into the controls, Validation Attributes comes into picture, display’s “Error Messages” and will not allow the page to be submitted to the server. By default, “Validation Error Messages” are displayed by using the “Span” tag and the text will be in black color, whereas if we want to highlight those messages go to “`<head>`” section of the “AddUser” View and define a style for “``” tag as following:

```
<style>span { color: red; background-color: yellow; font-style: oblique }</style>
```

Validation Attributes will perform Data Validations on the Client Machine and if at all those validations fail, page will not be submitted to server. Even in case of the “Remote” validation also without submitting the page to Server it will perform an “AJAX” call to the Server and receives the result as “JSON” which will be “true” if the validation is successful and “false” if the validation fails. In case, if Java Script is disabled on client’s browser, then there will be a problem i.e., page gets submitted to Server even if Validations fail, so to overcome the problem, in our “DisplayUser” action method we need to verify with a condition i.e., “ModelState.IsValid” which returns “true” if all the data validations are successfully completed or else it returns “false” if any validation fails.

To test this, re-write code of “DisplayUser” Action method as below:

```

if (ModelState.IsValid)
{
    return View(obj);
}
else
{
    return View("AddUser", obj);
}

```

All Validation Attributes will perform Validations on client side only and if at all Java Script is disabled on client’s browser, page will be submitted to the Server, so to overcome this problem, Server will again re-validate all the input, and if the Validations are successful, it will set the Models “IsValid” property as “true”, or “false” if any of the Validations fail which we can use in our Server-Side Logic to perform further actions as we have done above.

Now add a View to “DisplayUser” action method and to do this right click on the “DisplayUser” method in Controller class, select “Add View” and in the window opened choose the option “Details” under “Template DropDownList” and choose “User” under “Model Class DropDownList” which will create a View to display data and this is also “Scaffolding”. Run the “AddUser.cshtml”, fill in form with correct values and click on the “Submit” button which will launch “DisplayUser.cshtml” file, by displaying all the values we have entered in “AddUser” View.

If we open “DisplayUser.cshtml” file and watch the code, there we find a “**<dl> (Description List)**” to display the data and under this “**<dt>**” tag is used to display “Attribute Name” by calling the helper method “**DisplayNameFor**” and “**<dd>**” tag is used to display “Attribute Value” by calling the helper method “**DisplayFor**”.

Validation Summary: by using this we can display all the “Error Messages” at 1 place without displaying them beside the controls. To display all our “Error Messages” on top of the form, write the below code in “**AddUser**” View in the top of “**Begin Form**” block.

```
<div>@Html.ValidationSummary("", new { @class = "alert alert-danger" })</div>
```

Note: After adding this all the “Error Messages” will be displayed by the “**Validation Summary**” i.e., all at 1 place, but the “Error Messages” will be displayed beside the controls also and to avoid that, change the style code we have defined for “****” tag that we have added in “**<head>**” section as below:

Old Code: <style>span { color: red; background-color: yellow; font-style: oblique }</style>

New Code: <style>span { display: none }</style>

Language Integrated Query

In C# 3.0 .Net has introduced a new language known as "LINQ" much like "SQL (which we use universally with relational databases to perform queries)". LINQ allows you to write query expressions like SQL queries that can retrieve information from a wide variety of Data Sources like Objects, Relational Databases and XML.

Introduction to LINQ: LINQ stands for **Language Integrated Query**. LINQ is a data querying methodology which provides querying capabilities to .NET languages with syntax like a **SQL Query**. LINQ has a great power of querying from any source of data i.e., the **Data Source** can be collections of **Objects**, **Relational Databases** or **XML**.



LINQ to Objects: used to perform queries against the **in-memory** data like an **Array** or **Collection**.

LINQ to Databases:

- LINQ to DataSet's is used to perform queries against ADO.NET **Data Table's**.
- LINQ to SQL is used to perform queries against **Microsoft SQL Server** Database.
- LINQ to Entities is used to perform queries against any **Relation Database** like **SQL Server**, **Oracle**, etc.

LINQ to XML (XLinq): used to perform queries against the **XML Source**.

Advantages of LINQ:

- LINQ offers an **object-based**, language-integrated way to query over data, no matter where that data came from. So, through LINQ we can query **Database**, **XML** as well as **Collections** and **Arrays** also.
- **Compile time** syntax checking.
- It allows us to **Query Collections**, **Arrays**, & **Classes** etc. in the native language of our application like **VB** or **C#**.

LINQ to SQL

Probably the biggest and most exciting addition to the .Net Framework 3.5 is the addition of the .Net **Language Integrated Query Framework (LINQ)** into **C# 3.0**. Basically, what **LINQ** provides is a **lightweight façade** over programmatic data integration. This is such a big deal because **data is King**. Pretty much every application deal with data in some manner, whether that data comes from memory, databases, XML files, text files, or something else. Many developers find it very difficult to move from the strongly typed object-oriented world of C# to the data tier where objects are second-class citizens. The transition from the one world to the next was a kludge at best and was full of error-prone actions.

In C#, programming with objects means a wonderful strongly typed ability to work with code. You can navigate very easily through the namespaces; work with a debugger in the Visual Studio IDE, and more. However,

when you must access data, you will notice that things are dramatically different. You end up in a world that is not strongly typed, where debugging is a pain or even non-existent, and you end up spending most of the time sending strings to the database as commands. As a developer, you also must be aware of the underlying data and how it is.

Microsoft has provided LINQ as a lightweight façade that provides a strongly typed interface to the underlying data stores. LINQ provides the means for developers to stay within the coding environment they are used to and access the underlying data as objects that work with the IDE, Intellisense, and even debugging. With LINQ, the queries that you create now become first-class citizens within the .NET Framework alongside everything else you are used to. When you work with queries for the data store you are working with, you will quickly realize that they now work and behave as if they are types in the system. This means that you can now use any .NET-compliant language and query the underlying data stores as you never have before.

LINQ to SQL and Visual Studio: LINQ to SQL in particular is a means to have a strongly typed interface against a SQL Server Database. You will find the approach that LINQ to SQL provides is by far the easiest approach for querying SQL Server available now. It is important to remember that LINQ to SQL is not only about querying data, but you can also perform Insert/Update/Delete operations that you need to perform which are known as CRUD operations (Create/Read/Update/Delete). Visual Studio comes into strong play with LINQ to SQL in that you will find an extensive user interface that allows you to design the LINQ to SQL classes you will work with.

Adding a LINQ to SQL Class: To work with LINQ first you need to convert Relational Objects of Database into Object Oriented Types of our language and the process of conversion is known as **ORM (Object Relational Mapping)** and to perform this we are provided with a tool under Visual Studio i.e., OR Designer (Object-Relational Designer) which does an outstanding job of making it as easy as possible.

To work with **LINQ to SQL** first create a **Database in SQL Server** with the name “**MVCDB**” and then create a table with the name **Student** using the below **SQL Statement**:

Create Table Student (Sid Int Primary Key, Name Varchar(50), Class Int, Fees Money, Photo Varchar(100), Status Bit Not Null Default 1)

Now create a new “**ASP.NET Web Application**” project, naming it as “**MVCWithLinq1**”, choose “**MVC Project Template**” and click on **Create** button to create the Project. In the project created open **Solution Explorer** and under the **Models** folder add the “**OR Designer**” and to do this open “**Add New Item**” Window, in the LHS of the Window select “**Data**” and in the RHS select the option “**Linq to SQL Classes**” name the item as “**MVCDB.dbml**” (**Database Markup Language**). We can give any name to the “**.dbml**” file but it is always suggested to use our **Database name** as a name to this, because our database name is “**MVCDB**” we have named it as “**MVCDB.dbml**”, now click “**Add**” button which will do the following:

- Adds a reference to “**System.Data.Linq**” assembly which is required to work with “**LINQ to SQL**”.
- Under “**Models**” folder of Solution Explorer we will find “**MVCDB.dbml**” and under it we will find 2 sub-items “**MVCDB.dbml.layout**” and “**MVCDB.Designer.cs**” and under this file only “**OR-Designer**” writes all the “**ORM**” code by converting “**Relational Objects**” into “**Object Oriented Types**”.
- The item “**MVCDB.dbml**” is added to the studio which will appear as a tab within the document window, and this is made up of two parts. The first part on the left is for “**Data Classes**”, which map to **Tables**, **Views**, etc, dragging such items on this surface will give us a visual representation of those objects. The second part on the right is for **Methods**, which map to the **Stored Procedures** within the **Database**.

Let us have a look into the code of “`MVCDB.designer.cs`” file and there we will find a class “`MVCDBDataContext`” inheriting from “`DataContext`” class of “`System.Data.Linq`” namespace. “`DataContext`” class works with the “`Connection String`” and connects to the `Database` for any required operations when we create instance of the class. “`DataContext`” class also contains methods in it like “`CreateDatabase`”, “`DeleteDatabase`”, “`GetTable`”, “`ExecuteQuery`”, “`SubmitChanges`” etc, using which we can perform action directly on the `Database`. The “`MVCDBDataContext`” class which is defined here by default contains 4 parameterized `Constructors` in it.

Creating the Student Entity: For this example, we want to work with the “`Student`” (Entity) table from the “`MVCDB`” database, which means that you are going to create a “`Student`” (Entity) class that will use LINQ to SQL map to `Student` table. To accomplishing this task simply open the “`Server Explorer`” within `Visual Studio` from “`View`” menu and configure it with our Database i.e., “`MVCDB`”.

To configure the Database under “`Server Explorer`”, right click on “`Data Connections`” node and select the option “`Add Connection`” which opens a window “`Choose Data Source`” and in that select “`Microsoft SQL Server`” and click on “`OK`” button, which opens another window “`Add Connection`”, in the new window under “`Server name:`” `TextBox` enter your “`Server Name`”, under “`Log on to the server`” option choose the “`Authentication Mode`” and provide the “`Credentials`”, check the Checkbox “`Trust Server Certificate`”, and under “`Connect to a database`” option choose your “`Database`” from “`Select or enter a database name`” `DropDownList` (Database name is “`MVCDB`” in our case) and click on the “`OK`” button which adds the Database under “`Data Connections`” node.

Now drag and drop the “`Student`” table of our “`MVCDB`” Database onto the design surface of “`O/R Designer`” in to `LHS` and this action will add a bunch of code in to “`MVCDB.designer.cs`” file on our behalf with a set of classes in it, and those classes will give you a strongly typed access to the `Student` (Entity) table.

When we drag & drop the first object (Table or SP) on OR Designer it will perform the following actions:

- Writes `Connection String` in to “`Web.config`” file targeting to the `Database` we have configured in `Server Explorer`.
- Defines a new default constructor in “`MVCDBDataContext`” class and we can use this `constructor` for creating the instance of “`MVCDBDataContext`” class for connecting to the `Database` and this constructor will read the `Connection String` from “`Web.config`” file whereas in the latest version this `default Constructor` is not defined, so we need to use the existing parameterized constructor of “`MVCDBDataContext`” class that takes `string` as a `parameter` and for that we need to pass the `name of Connection String` by reading it from “`Web.config`” file.

When we drag & drop a table on OR Designer the following actions gets performed internally:

- Defines a class representing the `Entity (Table)` we have dragged & dropped on the `OR Designer` where the name of the class will be same as the table name, right now we have dropped “`Student`” table on `OR Designer`, so “`Student`” class gets defined.
- Defines properties under the entity class (`Student`), where each property represents each attribute of the entity (table). So, in the student class we will find the properties `Sid`, `Name`, `Class`, `Fees`, `Photo` and `Status`.
- Defines a property under “`MVCDBDataContext`” class referring to the table we are working with, and the type of this property will be “`Table<Entity>`”, because we are working with “`Student Entity`” here the name of the property will be “`Students`” and type of the property will be “`Table<Student>`”.

Note: `Table<Entity>` is a generic class under “`System.Data.Linq`” namespace which also contains a set of methods like “`DeleteOnSubmit`”, “`InsertOnSubmit`”, “`SingleOrDefault`”, “`FirstOrDefault`”, “`Single`”, “`First`” etc. for performing `CRUD` operations on the `Entity`.

Performing CRUD (Create (Insert), Read (Select), Update and Delete) Operations: to Perform CRUD operations on SQL Server Database by using Linq to SQL we need to adopt the below process for Insert, Update and Delete:

Steps for Inserting:

1. Create a instance of Model (**Student**) class, which is defined representing the **Student Entity (Table)** because each **instance** is a record, and then assign values to **properties** because those **properties** represent **attributes**.
2. Call **InsertOnSubmit** method on the table (**Students**) which adds the record into the table in a **pending state**.
3. Call **SubmitChanges** method on **DataContext** object for saving the changes to **Database** server.

Steps for Updating:

1. Create a reference of Model (**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 **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 Model (**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.

Add a class under Models folder with the name StudentDAL.cs and write the below code in it:

```
using System.Configuration;
public class StudentDAL
{
    MVCDBDataContext context = new MVCDBDataContext(); //Old Version (If default constructor is defined)
    Or
    MVCDBDataContext context = new MVCDBDataContext(
        ConfigurationManager.ConnectionStrings["MVCDBConnectionString"].ConnectionString); //New Version

    public List<Student> GetStudents(bool? Status)
    {
        List<Student> students;
        try
        {
            if(Status != null)
                students = (from s in context.Students where s.Status == Status select s).ToList();
            else
                students = context.Students.ToList(); Or students = (from s in context.Students select s).ToList();
            return students;
        }
        catch(Exception ex)
        {
            throw ex;
        }
    }
}
```

```

public Student GetStudent(int Sid, bool? Status)
{
    Student student;
    try
    {
        if (Status == null)
            student = (from s in context.Students where s.Sid == Sid select s).Single();
        else
            student = (from s in context.Students where s.Sid == Sid && s.Status == Status select s).Single();
        return student;
    }
    catch(Exception ex)
    {
        throw ex;
    }
}
public void AddStudent(Student student)
{
    try
    {
        context.Students.InsertOnSubmit(student);
        context.SubmitChanges();
    }
    catch(Exception ex)
    {
        throw ex;
    }
}
public void UpdateStudent(Student newValues)
{
    try
    {
        Student oldValues = context.Students.Single(s => s.Sid == newValues.Sid);
        oldValues.Name = newValues.Name;
        oldValues.Class = newValues.Class;
        oldValues.Fees = newValues.Fees;
        oldValues.Photo = newValues.Photo;
        context.SubmitChanges();
    }
    catch(Exception ex)
    {
        throw ex;
    }
}
public void DeleteStudent(int Sid)
{

```

```

try
{
    Student oldValues = context.Students.First(S => S.Sid == Sid);
    //dc.Students.DeleteOnSubmit(oldValues);           //Permenant Deletion
    oldValues.Status = false;                         //Updates the status with-out deleting the record
    context.SubmitChanges();
}
catch (Exception ex)
{
    throw ex;
}
}
}

```

Add a Controller under the Controllers folder with the name StudentController and write the below code in it:

```

using System.IO;
using MVCWithLinq1.Models;
public class StudentController : Controller
{
    StudentDAL obj = new StudentDAL();
    public ViewResult DisplayStudents()
    {
        return View(obj.GetStudents(true));
    }
    public ViewResult DisplayStudent(int Sid)
    {
        return View(obj.GetStudent(Sid, true));
    }
    [HttpGet]
    public ViewResult AddStudent()
    {
        return View(new Student());
    }
    [HttpPost]
    public RedirectToRouteResult AddStudent(Student student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            //Checking whether the folder "Uploads" is exists or not and creating it if not exists
            string PhysicalPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(PhysicalPath))
            {
                Directory.CreateDirectory(PhysicalPath);
            }
            selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
            student.Photo = selectedFile.FileName;
        }
    }
}

```

```

student.Status = true;
obj.AddStudent(student);
return RedirectToAction("DisplayStudents");
}
public ViewResult EditStudent(int Sid)
{
    Student student = obj.GetStudent(Sid, true);
    TempData["Photo"] = student.Photo;
    return View(student);
}
public RedirectToRouteResult UpdateStudent(Student student, HttpPostedFileBase selectedFile)
{
    if (selectedFile != null)
    {
        string PhysicalPath = Server.MapPath("~/Uploads/");
        if (!Directory.Exists(PhysicalPath))
        {
            Directory.CreateDirectory(PhysicalPath);
        }
        selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
        student.Photo = selectedFile.FileName;
    }
    else if(TempData["Photo"] != null)
    {
        student.Photo = TempData["Photo"].ToString();
    }
    obj.UpdateStudent(student);
    return RedirectToAction("DisplayStudents");
}
public RedirectToRouteResult DeleteStudent(int Sid)
{
    obj.DeleteStudent(Sid);
    return RedirectToAction("DisplayStudents");
}
}

```

Add a view with the name DisplayStudents.cshtml, selecting layout Checkbox and write the below code in it:

```

@model IEnumerable<MVCWithLinq1.Models.Student>


## Student Details



| @Html.DisplayNameFor(s => s.Sid) | @Html.DisplayNameFor(s => s.Name) | @Html.DisplayNameFor(s => s.Class) | @Html.DisplayNameFor(s => s.Fees) | @Html.DisplayNameFor(s => s.Photo) | Actions |
|----------------------------------|-----------------------------------|------------------------------------|-----------------------------------|------------------------------------|---------|
|----------------------------------|-----------------------------------|------------------------------------|-----------------------------------|------------------------------------|---------|


```

```

</tr>
@foreach (MVCWithLinq1.Models.Student student in Model)
{
<tr>
<td>@Html.DisplayFor(s => student.Sid)</td>
<td>@Html.DisplayFor(s => student.Name)</td>
<td>@Html.DisplayFor(s => student.Class)</td>
<td>@Html.DisplayFor(s => student.Fees)</td>
<td><img src='/Uploads/@student.Photo' width='40' height='25' alt='No Image' /></td>
<td>
    @Html.ActionLink("View", "DisplayStudent", new { Sid = student.Sid })
    @Html.ActionLink("Edit", "EditStudent", new { Sid = student.Sid })
    @Html.ActionLink("Delete", "DeleteStudent", new { Sid = student.Sid },
        new { onclick = "return confirm('Are you sure of deleting the record?')" })
</td>
</tr>
}
<tr><td colspan="6" align="center">@Html.ActionLink("Add New Student", "AddStudent")</td></tr>
</table>

```

Add a view with the name DisplayStudent.cshtml, selecting layout Checkbox and write the below code in it:

```

@model MVCWithLinq1.Models.Student
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center">
<tr>
<td rowspan="4"><img src='~/Uploads/@Model.Photo' width="200" height="200" alt="No Image" /></td>
<td>Sid: @Model.Sid</td>
</tr>
<tr><td>Name: @Model.Name</td></tr>
<tr><td>Class: @Model.Class</td></tr>
<tr><td>Fees: @Model.Fees</td></tr>
<tr>
<td colspan="2" align="center">@Html.ActionLink("Back to Student Details", "DisplayStudents")</td>
</tr>
</table>

```

Add a view with the name AddStudent.cshtml, selecting layout Checkbox and write the below code in it:

```

@model MVCWithLinq1.Models.Student
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
@using (Html.BeginForm("AddStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
<div>@Html.LabelFor(s => s.Sid)<br />@Html.TextBoxFor(s => s.Sid)</div>
<div>@Html.LabelFor(s => s.Name)<br />@Html.TextBoxFor(s => s.Name)</div>
<div>@Html.LabelFor(s => s.Class)<br />@Html.TextBoxFor(s => s.Class)</div>
<div>@Html.LabelFor(s => s.Fees)<br />@Html.TextBoxFor(s => s.Fees)</div>
<div>@Html.LabelFor(s => s.Photo)<br /><input type="file" name="selectedFile" /></div>
<div>
<input type="submit" value="Save" />

```

```

<input type="reset" value="Reset" />
</div>
}
@Html.ActionLink("Back to Student Details", "DisplayStudents")


---



Add a view with the name EditStudent.cshtml, selecting layout Checkbox and write the below code in it:



```

@model MVCWithLinq1.Models.Student
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Student</h2>
@using (Html.BeginForm("UpdateStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
 <div>@Html.LabelFor(S => S.Sid)
@Html.TextBoxFor(S => S.Sid)</div>
 <div>@Html.LabelFor(S => S.Name)
@Html.TextBoxFor(S => S.Name)</div>
 <div>@Html.LabelFor(S => S.Class)
@Html.TextBoxFor(S => S.Class)</div>
 <div>@Html.LabelFor(S => S.Fees)
@Html.TextBoxFor(S => S.Fees)</div>
 <div>
 @Html.LabelFor(S => S.Photo)

 <input type="file" name="selectedFile" />
 </div>
 <div>
 <input type="submit" value="Update" />
 @Html.ActionLink("Cancel", "DisplayStudents")
 </div>
}

```


```

Calling Stored Procedures using LINQ

If we want to call **Stored Procedures** of **SQL Server Database** by using **LINQ** we need to first **drag & drop** those **Procedures** on to the **RHS panel of OR-Designer** from **Server Explorer**, so that they get converted into methods under our “**DataContext**” class i.e., “**MVCDBDataContext**” with the same name of **Procedures**. In this process if the **Stored Procedure** has any parameters, then those parameters will be defined for **Method** also, i.e., “**input**” parameters of **Procedure** will become “**input**” parameters of the **Method** and “**output**” parameters of the **Procedure** will become “**ref**” parameters of the **Method**.

For example, if the below SP was dropped on RHS panel of OR-designer:

```
Create Procedure Add(@x int, @y int, @z int output)
```

A method gets defined under **MVCDBDataContext class as following:**

```
public int Add(int? x, int? y, ref int? z)
```

If the **Procedure** is defined to perform any **Non-Query** operations then **return type** of the method will be **integer**, whereas if the **Procedure** is defined to perform **Query** operations then **return type** of the method will be “**ISingleResult<T>**”, where “**T**” represents a class that is newly defined when we **drag & drop** the **Query Procedure** and the name of the new class will be “**SP Name**” suffixed with “**Result**” i.e., for example if the **Procedure** name is “**Student_Select**” then the class name will be “**Student_SelectResult**”.

To test calling **Stored Procedures**, create a new “**ASP.NET Web Application**” project, naming it as “**MVCWithLinq2**” choose “**MVC Project Template**” and click on “**Create**” button. Now open “**Solution Explorer**” and under the **Models** folder add “**OR Designer**” and to do that open “**Add New Item**” window, select the option “**Linq**

to SQL Classes”, name it as “**MVCDB.dbml**” and click on the **Add** button. Now open “**Server Explorer**” and configure our “**MVCDB**” Database under it and create the below 4 **Stored Procedure’s** in the **Database** to perform **CRUD** Operations on “**Student**” table. To create a Procedure right click on the “**Stored Procedures**” node under the **Database** and select the option “**Add New Stored Procedure**” which opens a window, write the **Procedure** code over there and after implementing the code write click on the document window and select “**Execute**” which will create the **Procedure** in the **Database**.

```

CREATE Procedure Student_Select(@Sid Int=NULL, @Status Bit=NULL)
As
Begin
    If @Sid Is Null And @Status Is Null          --Fetches all the records of table
        Select Sid, Name, Class, Fees, Photo From Student Order By Sid;
    Else If @Sid Is Null And @Status Is Not Null   --Fetches records based on Status
        Select Sid, Name, Class, Fees, Photo From Student Where Status=@Status Order By Sid;
    Else If @Sid Is Not Null And @Status Is Null     --Fetches a single record based on Sid
        Select Sid, Name, Class, Fees, Photo From Student Where Sid=@Sid;
    Else If @Sid Is Not Null And @Status Is Not Null  --Fetches a single record based on Sid & Status
        Select Sid, Name, Class, Fees, Photo From Student Where Sid=@Sid And Status=@Status;
End;


---


Create Procedure Student_Insert(@Sid int, @Name varchar(50), @Class int, @Fees money, @Photo varchar(100)=null)
As
    Insert Into Student (Sid, Name, Class, Fees, Photo) Values (@Sid, @Name, @Class, @Fees, @Photo)


---


Create Procedure Student_Update(@Sid int, @Name varchar(50), @Class int, @Fees money, @Photo varchar(100)=Null)
As
    Update Student Set Name=@Name, Class=@Class, Fees=@Fees, Photo=@Photo Where Sid=@Sid;


---


Create Procedure Student_Delete(@Sid int)
As
    Update Student Set Status=0 Where Sid=@Sid;

```

Now drag and drop the 4 **Stored Procedures** we have defined above on to the **RHS Panel** of “**OR Designer**”, which will generate the required methods in “**MVCDBDataContext**” class. It also generates a class with the name “**Student_SelectResult**” mapping with the results retrieved by “**Student_Select**” Stored Procedure.

Note: In this case we don’t have “**Student (Model)**” class generated because we did not drag and drop “**Student**” table on to the “**OR-Designer**”, so use “**Student_SelectResult**” class as your **Model** to perform **Model Binding** with the **Views**.

Add a controller in Controllers folder naming it as **StudentController.cs and write the below code in it:**

```

using System.IO;
using System.Configuration;
using MVCWithLinq2.Models;
public class StudentController : Controller
{
    MVCDBDataContext context = new MVCDBDataContext(); //Old Version (If default constructor is defined)
    Or
    MVCDBDataContext context = new MVCDBDataContext(
        ConfigurationManager.ConnectionStrings["MVCDBConnectionString"].ConnectionString); //New Version

```

```

public ViewResult DisplayStudents()
{
    List<Student_SelectResult> students = dc.Student_Select(null, true).ToList();
    return View(students);
}
public ViewResult DisplayStudent(int Sid)
{
    Student_SelectResult student = dc.Student_Select(Sid, true).ToList()[0];
    return View(student);
}
[HttpGet]
public ViewResult AddStudent()
{
    Student_SelectResult student = new Student_SelectResult();
    return View(student);
}
[HttpPost]
public RedirectToRouteResult AddStudent(Student_SelectResult student, HttpPostedFileBase selectedFile)
{
    if(selectedFile != null)
    {
        string folderPath = Server.MapPath("~/Uploads/");
        if(!Directory.Exists(folderPath))
        {
            Directory.CreateDirectory(folderPath);
        }
        selectedFile.SaveAs(folderPath + selectedFile.FileName);
        student.Photo = selectedFile.FileName;
    }
    dc.Student_Insert(student.Sid, student.Name, student.Class, student.Fees, student.Photo);
    return RedirectToAction("DisplayStudents");
}
public ViewResult EditStudent(int Sid)
{
    Student_SelectResult student = dc.Student_Select(Sid, true).ToList()[0];
    TempData["Photo"] = student.Photo;
    return View(student);
}
public RedirectToRouteResult UpdateStudent(Student_SelectResult student, HttpPostedFileBase selectedFile)
{
    if (selectedFile != null)
    {
        string folderPath = Server.MapPath("~/Uploads/");
        if (!Directory.Exists(folderPath))
        {
            Directory.CreateDirectory(folderPath);
        }
    }
}

```

```

        }
        selectedFile.SaveAs(folderPath + selectedFile.FileName);
        student.Photo = selectedFile.FileName;
    }
    else if(TempData["Photo"] != null) {
        student.Photo = TempData["Photo"].ToString();
    }
    dc.Student_Update(student.Sid, student.Name, student.Class, student.Fees, student.Photo);
    return RedirectToAction("DisplayStudents");
}
public RedirectToRouteResult DeleteStudent(int Sid)
{
    dc.Student_Delete(Sid);
    return RedirectToAction("DisplayStudents");
}
}

```

Add a view with the name DisplayStudents.cshtml, selecting layout Checkbox and write the below code in it:

```

@model IEnumerable<MVCWithLinq2.Models.Student_SelectResult>
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Students Details</h2>
<table border="1" align="center" class="table-condensed">
<tr>
    <th>@Html.DisplayNameFor(S => S.Sid)</th>
    <th>@Html.DisplayNameFor(S => S.Name)</th>
    <th>@Html.DisplayNameFor(S => S.Class)</th>
    <th>@Html.DisplayNameFor(S => S.Fees)</th>
    <th>@Html.DisplayNameFor(S => S.Photo)</th>
    <th>Actions</th>
</tr>
@foreach (MVCWithLinq2.Models.Student_SelectResult student in Model)
{
    <tr>
        <td>@Html.DisplayFor(S => student.Sid)</td>
        <td>@Html.DisplayFor(S => student.Name)</td>
        <td>@Html.DisplayFor(S => student.Class)</td>
        <td>@Html.DisplayFor(S => student.Fees)</td>
        <td><img src='/Uploads/@student.Photo' width="40" height="25" alt="No Image" /></td>
        <td>
            @Html.ActionLink("View", "DisplayStudent", new { Sid = student.Sid })
            @Html.ActionLink("Edit", "EditStudent", new { Sid = student.Sid })
            @Html.ActionLink("Delete", "DeleteStudent", new { Sid = student.Sid },
                new { onclick = "return confirm('Are you sure of deleting the record?')" })
        </td>
    </tr>
}
<tr><td colspan="6" align="center">@Html.ActionLink("Add New Student", "AddStudent")</td></tr>
</table>

```

Add a view with the name DisplayStudent.cshtml, selecting layout Checkbox and write the below code in it:

```
@model MVCWithLinq2.Models.Student_SelectResult
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center">
<tr>
<td rowspan=4><img src='~/Uploads/@Model.Photo' width="200" height="200" alt="No Image" /></td>
<td>Sid: @Model.Sid</td>
</tr>
<tr><td>Name: @Model.Name</td></tr>
<tr><td>Class: @Model.Class</td></tr>
<tr><td>Fees: @Model.Fees</td></tr>
<tr><td colspan="2" align="center">@Html.ActionLink("Back to Student Details", "DisplayStudents")</td></tr>
</table>
```

Add a view with the name AddStudent.cshtml, selecting layout Checkbox and write the below code in it:

```
@model MVCWithLinq2.Models.Student_SelectResult
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
@using (Html.BeginForm("AddStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid)</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
    <div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
    <div>@Html.LabelFor(S => S.Photo)<br /><input type="file" name="selectedFile" /></div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" name="btnReset" />
    </div>
}
@Html.ActionLink("Back to Student Details", "DisplayStudents")
```

Add a view with the name EditStudent.cshtml, selecting layout Checkbox and write the below code in it:

```
@model MVCWithLinq2.Models.Student_SelectResult
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Student Details</h2>
@using (Html.BeginForm("UpdateStudent", "Student", FormMethod.Post, new { enctype="multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid, new { @readonly = "true" })</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
    <div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
    <div>
        @Html.LabelFor(S => S.Photo)
        <br />
        <img src='~/Uploads/@Model.Photo' width="50" height="50" alt="No Image" style="border:dashed 2px red" />
        <input type="file" name="selectedFile" />
    </div>
    <div>
        <input type="submit" value="Update" name="btnUpdate" />
    </div>
}
```

```
    @Html.ActionLink("Cancel", "DisplayStudents")
</div>
}
```

Working with multiple tables using LINQ to SQL

We can work with multiple tables that are present under **Database** which have a relationship with each other by using **LINQ**, but to do that we need to define a **model** class that contains attributes of both the **Entities**. To test this, create 2 new tables under our “**MVCDB**” Database with the name “**Department**” and “**Employee**”, and insert few records into “**Department**” table as following:

```
➤ Create Table Department(Did Int Constraint Did_PK Primary Key Identity(10, 10), Dname Varchar(50), Location  
      Varchar(50));  
➤ Insert Into Department Values ('Marketing ', 'Mumbai ');  
➤ Insert Into Department Values ('Sales ', 'Chennai ');  
➤ Insert Into Department Values ('Accounting ', 'Hyderabad ');  
➤ Insert Into Department Values ('Finance ', 'Delhi ');  
➤ Create Table Employee(Eid Int Constraint Eid_PK Primary Key Identity(1001, 1), Ename Varchar(50), Job  
      Varchar(50), Salary Money, Did Int Constraint Did_FK References Department(Did), Status Bit Not Null default 1);
```

To test working with multiple tables, create a new “**ASP.NET Web Application**” project, naming it as “**MVCWithLinq3**”, choose “**MVC Project Template**” and click on the “**Create**” button. Now under the **Models** folder add “**OR Designer**” naming it as “**MVCDB.dbml**”. Configure our “**MVCDB**” Database under “**Server Explorer**” and then drag & drop “**Department**” and “**Employee**” Tables on the **LHS Panel** of **OR Designer**. Now under **Models** folder add a **Class** with the name “**EmpDept.cs**” that combines the properties of **Employee** & **Department** model classes, and write the below code in it:

```
using System.Web.Mvc;
public class EmpDept
{
    public int Eid { get; set; }
    public string Ename { get; set; }
    public string Job { get; set; }
    public decimal? Salary { get; set; }
    public int Did { get; set; }
    public string Dname { get; set; }
    public string Location { get; set; }
    public List<SelectListItem> Departments { get; set; }
}
```

Add another class under Models folder with the name EmployeeDAL.cs and write the below code in it:

```
using System.Web.Mvc;
public class EmployeeDAL
{
    MVCDBDataContext dc = new MVCDBDataContext();
    public List<SelectListItem> GetDepartments()
    {
        List<SelectListItem> Depts = new List<SelectListItem>();
```

```

foreach (var Item in dc.Departments)
{
    SelectListItem li = new SelectListItem { Text = Item.Dname, Value = Item.Did.ToString() };
    Depts.Add(li);
}
return Depts;
}

public EmpDept GetEmployee(int Eid)
{
    var Record = (from E in dc.Employees join D in dc.Departments on E.Did equals D.Did where E.Eid == Eid select
        new { E.Eid, E.Ename, E.Job, E.Salary, D.Did, D.Dname, D.Location }).Single();
    EmpDept Emp = new EmpDept { Eid = Record.Eid, Ename = Record.Ename, Job = Record.Job,
        Salary = Record.Salary, Did = Record.Did, Dname = Record.Dname, Location = Record.Location };
    return Emp;
}

public List<EmpDept> GetEmployees()
{
    var Records = from E in dc.Employees join D in dc.Departments on E.Did equals D.Did where E.Status == true
        select new { E.Eid, E.Ename, E.Job, E.Salary, D.Did, D.Dname, D.Location };
    List<EmpDept> Emps = new List<EmpDept>();
    foreach (var Record in Records)
    {
        EmpDept Emp = new EmpDept { Eid = Record.Eid, Ename = Record.Ename, Job = Record.Job,
            Salary = Record.Salary, Did = Record.Did, Dname = Record.Dname, Location = Record.Location };
        Emps.Add(Emp);
    }
    return Emps;
}

public void Employee_Insert(EmpDept obj)
{
    Employee Emp = new Employee { Ename = obj.Ename, Job = obj.Job, Salary = obj.Salary, Did = obj.Did,
        Status = true };
    dc.Employees.InsertOnSubmit(Emp);
    dc.SubmitChanges();
}

public void Employee_Update(EmpDept NewValues)
{
    Employee OldValues = dc.Employees.Single(E => E.Eid == NewValues.Eid);
    OldValues.Ename = NewValues.Ename;
    OldValues.Job = NewValues.Job;
    OldValues.Salary = NewValues.Salary;
    OldValues.Did = NewValues.Did;
    dc.SubmitChanges();
}

public void Employee_Delete(int Eid)
{
}

```

```

Employee OldValues = dc.Employees.Single(E => E.Eid == Eid);
OldValues.Status = false;
dc.SubmitChanges();
}
}

```

Add a Controller under Controllers folder with the name EmployeeController.cs and write the below code in it:

```

using MVCWithLinq3.Models;
public class EmployeeController : Controller
{
    EmployeeDAL obj = new EmployeeDAL();
    public ViewResult DisplayEmployees()
    {
        return View(obj.GetEmployees());
    }
    public ViewResult DisplayEmployee(int eid)
    {
        return View(obj.GetEmployee(eid));
    }
    [HttpGet]
    public ViewResult AddEmployee()
    {
        EmpDept emp = new EmpDept();
        emp.Departments = obj.GetDepartments();
        return View(emp);
    }
    [HttpPost]
    public RedirectToRouteResult AddEmployee(EmpDept emp)
    {
        obj.Employee_Insert(emp);
        return RedirectToAction("DisplayEmployees");
    }
    public ViewResult EditEmployee(int eid)
    {
        EmpDept Emp = obj.GetEmployee(eid);
        Emp.Departments = obj.GetDepartments();
        return View(Emp);
    }
    public RedirectToRouteResult UpdateEmployee(EmpDept emp)
    {
        obj.Employee_Update(emp);
        return RedirectToAction("DisplayEmployees");
    }
    public RedirectToRouteResult DeleteEmployee(int eid)
    {
        obj.Employee_Delete(eid);
    }
}

```

```

        return RedirectToAction("DisplayEmployee");
    }
}

```

Add a view with the name DisplayEmployees.cshtml, selecting layout Checkbox and write the below code in it:

```

@model IEnumerable<MVCWithLinq3.Models.EmpDept>
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Employee Details</h2>
<table border="1" align="center">
<tr>
    <th>@Html.DisplayNameFor(E => E.Eid)</th>
    <th>@Html.DisplayNameFor(E => E.Ename)</th>
    <th>@Html.DisplayNameFor(E => E.Job)</th>
    <th>@Html.DisplayNameFor(E => E.Salary)</th>
    <th>@Html.DisplayNameFor(E => E.Did)</th>
    <th>@Html.DisplayNameFor(E => E.Dname)</th>
    <th>@Html.DisplayNameFor(E => E.Location)</th>
    <th>Actions</th>
</tr>
@foreach (var Employee in Model)
{
    <tr>
        <td>@Html.DisplayFor(E => Employee.Eid)</td>
        <td>@Html.DisplayFor(E => Employee.Ename)</td>
        <td>@Html.DisplayFor(E => Employee.Job)</td>
        <td>@Html.DisplayFor(E => Employee.Salary)</td>
        <td>@Html.DisplayFor(E => Employee.Did)</td>
        <td>@Html.DisplayFor(E => Employee.Dname)</td>
        <td>@Html.DisplayFor(E => Employee.Location)</td>
        <td>
            @Html.ActionLink("View", "DisplayEmployee", new { Eid = Employee.Eid })
            @Html.ActionLink("Edit", "EditEmployee", new { Eid = Employee.Eid })
            @Html.ActionLink("Delete", "DeleteEmployee", new { Eid = Employee.Eid },
                new { onclick = "return confirm('Are you sure of deleting the record?') })
        </td>
    </tr>
}
<tr><td colspan="8" align="center">@Html.ActionLink("Add New Employee", "AddEmployee")</td></tr>
</table>

```

Add a view with the name DisplayEmployee.cshtml, selecting layout Checkbox and write the below code in it:

```

@model MVCWithLinq3.Models.EmpDept
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Employee Details</h2>
<table border="1" align="center">
<tr><td>Eid:</td><td>@Model.Eid</td></tr>
<tr><td>Ename:</td><td>@Model.Ename</td></tr>
<tr><td>Job:</td><td>@Model.Job</td></tr>

```



```

<tr><td>Salary:</td><td>@Model.Salary</td></tr>
<tr><td>Did:</td><td>@Model.Did</td></tr>
<tr><td>Dname:</td><td>@Model.Dname</td></tr>
<tr><td>Location:</td><td>@Model.Location</td></tr>
<tr>
    <td colspan="2" align="center">@Html.ActionLink("Back to Employee Details", "DisplayEmployees")</td>
</tr>
</table>

```

Add a view with the name AddEmployee.cshtml, selecting layout Checkbox and write the below code in it:

```

@model MVCWithLinq3.Models.EmpDept
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Employee</h2>
@using (Html.BeginForm("AddEmployee", "Employee"))
{
    <div>@Html.LabelFor(E => E.Ename)<br />@Html.TextBoxFor(E => E.Ename)</div>
    <div>@Html.LabelFor(E => E.Job)<br />@Html.TextBoxFor(E => E.Job)</div>
    <div>@Html.LabelFor(E => E.Salary)<br />@Html.TextBoxFor(E => E.Salary)</div>
    <div>
        @Html.LabelFor(E => E.Dname)<br />
        @Html.DropDownListFor(E => E.Did, Model.Departments, "-Select Dept-")
    </div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" name="btnReset" />
    </div>
}
@Html.ActionLink("Back to Display Employees", "DisplayEmployees")

```

Add a view with the name EditEmployee.cshtml, selecting layout Checkbox and write the below code in it:

```

@model MVCWithLinq3.Models.EmpDept
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Employee Details</h2>
@using (Html.BeginForm("UpdateEmployee", "Employee"))
{
    <div>@Html.LabelFor(E => E.Eid)<br />@Html.TextBoxFor(E => E.Eid, new { @readonly = "true" })</div>
    <div>@Html.LabelFor(E => E.Ename)<br />@Html.TextBoxFor(E => E.Ename)</div>
    <div>@Html.LabelFor(E => E.Job)<br />@Html.TextBoxFor(E => E.Job)</div>
    <div>@Html.LabelFor(E => E.Salary)<br />@Html.TextBoxFor(E => E.Salary)</div>
    <div>@Html.LabelFor(E => E.Dname)<br />@Html.DropDownListFor(E => E.Did, Model.Departments)</div>
    <div>
        <input type="submit" value="Update" name="btnUpdate" />
        @Html.ActionLink("Cancel", "DisplayEmployees")
    </div>
}

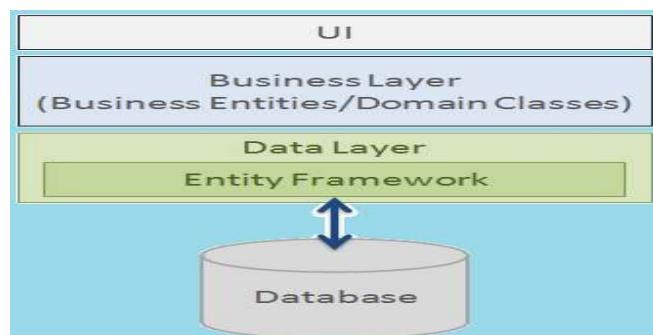
```

Note: while working with multiple tables, model binding is done with the class we have defined representing the 2 tables i.e., “**EmpDept**”, whereas while performing **CRUD** operations we will be working with the original **Model** class only i.e., “**Employee**” or “**Department**”.

Entity Framework

Prior to .NET 3.5, we (developers) often used to write “ADO.NET” code to save or retrieve application data from the underlying Database. We used to open a connection to the Database, create a “DataSet” or “DataReader” to fetch or submit the data to the Database and in this process, we convert data present in the “DataSet” or “DataReader” to .NET objects or vice-versa to apply business rules. This was a cumbersome and error prone process. Microsoft has provided a framework called “Entity Framework” to automate all these Database related activities for your application.

Entity Framework is an open source “ORM Framework” for .NET Applications supported by Microsoft. It enables developers to work with data using objects of domain specific classes without focusing on the underlying Database or Tables and Columns where this data is stored. With Entity Framework, developers can work at a higher level of abstraction when they deal with data and can create and maintain data-oriented applications with less code compared with traditional applications. “Entity Framework is an Object-Relational Mapper (O/RM) that enables .NET developers to work with a Database using .NET Objects. It eliminates the need for most of the data-access code that developers usually need to write.”



As per the above figure, Entity Framework fits between the business entities (domain classes) and the Database. It saves data stored in the properties of business entities and retrieves data from the Database and converts it to business entity objects automatically.

Entity Framework Versions: Microsoft introduced Entity Framework in 2008 with .NET Framework 3.5 SP1. Since then, it released many versions of Entity Framework. Currently, there are two latest versions of Entity Framework: “EF 6” and “EF Core”. The following table lists important difference between EF 6 and EF Core.

EF 6	EF Core
✓ First released in 2008 with .NET Framework 3.5 SP1	✓ First released in June 2016 with .NET Core 1.0
✓ Stable and feature rich	✓ New and evolving
✓ Windows only	✓ Windows, Linux, OSX
✓ Works on .NET Framework 3.5+	✓ Works on .NET Framework 4.5+ and .NET Core
✓ Open-source	✓ Open-source

EF 6 Version History:

EF 3.5,
EF 4.0, EF 4.1, EF 4.1.1, EF 4.2, EF 4.3, EF 4.3.1,
EF 5.0,
EF 6.0, EF 6.0.1, EF 6.0.2, EF 6.1.0, EF 6.1.1, EF 6.1.2, EF 6.1.3, EF 6.2.0, EF 6.3.0, EF 6.4.0, EF 6.4.4 (Current & Last)

EF Core Version History:

EF Core 1.0, EF Core 1.1
EF Core 2.0, EF Core 2.1, EF Core 2.2
EF Core 3.0, EF Core 3.1
EF Core 5.0, EF Core 6.0, EF Core 7.0, EF Core 8.0

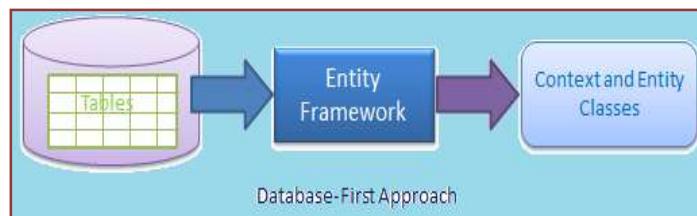
Entity Framework Features:

- **Cross-platform:** EF Core is a cross-platform framework which can run on Windows, Linux, and Mac.
- **Modelling:** EF (Entity Framework) creates an **EDM** (Entity Data Model) based on **POCO** (Plain Old CLR Object) entities with **get/set properties** of different data types. It uses this model when querying or saving entity data to the underlying **Database**.
- **Querying:** EF allows us to use **LINQ** queries to retrieve data from the underlying database. The database provider will translate this **LINQ** queries to the database-specific query language (e.g., **SQL** for a relational database). EF also allows us to execute raw **SQL** queries directly to the **database**.
- **Change Tracking:** EF keeps track of changes occurred to instances of your **entities** (Property values) which need to be submitted to the **Database**.
- **Saving:** EF executes **INSERT**, **UPDATE**, and **DELETE** commands to the **database** based on the changes occurred to your **entities** when you call the **SaveChanges()** method.
- **Concurrency:** EF uses **Optimistic Concurrency** by default to protect overwriting changes made by another user since data was fetched from the **database**.
- **Transactions:** EF performs **automatic transaction management** while querying or saving data. It also provides options to **customize transaction management**.
- **Caching:** EF includes first level of caching out of the box. So, repeated querying will return data from the cache instead of hitting the **Database**.
- **Built-in Conventions:** EF follows conventions over the configuration programming pattern and includes a set of default rules which automatically configure the **EF** model.
- **Configurations:** EF allows us to configure the **EF** model by using **Data Annotation** attributes or **Fluent API** to override default conventions.
- **Migrations:** EF provides a set of migration commands that can be executed on the **NuGet Package Manager Console** or the **Command Line Interface** to create or manage underlying **database Schema**.

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

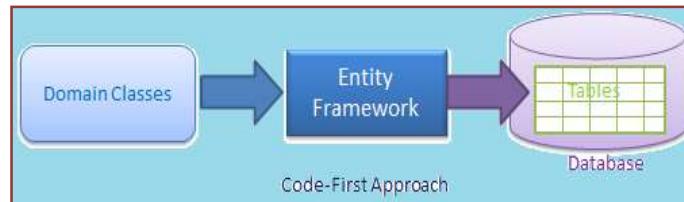
- **Database-First**
- **Code-First**
- **Model-First**

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**.

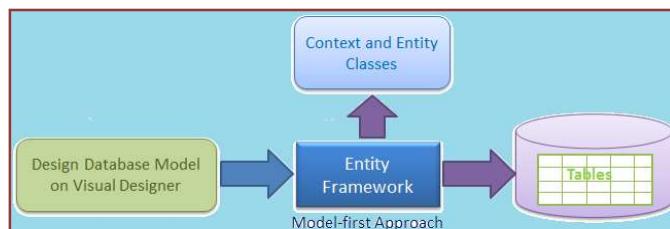


Note: EF 6 supports the **database-first** approach **extensively** whereas EF Core includes **limited support** for this.

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 generate 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, 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.

Choosing a Development Approach for our Application: use the following flow chart to decide which will be the right approach to develop an application using **Entity Framework**:



Database First Approach

To work with Entity Framework, create a new "**ASP.NET Web Application**" project naming it as "**MVCWithEFDBF1**", choose "**MVC Project Template**" and click on "**Create**" button. Now open Solution Explorer, right click on the Models folder and choose "**Add => New Item**" option and in the window opened select the item "**ADO.NET Entity Data Model**", name it as "**TestEF.edmx**", click "**Ok**" which opens a wizard and in that select the option "**EF Designer from database**" => click "**Next**" and in the window opened click on "**New Connection**" button and enter the connection details for our "**MVCDB**" Database and click "**Ok**" button which will close the "**New Connection**" window, now select the radio button "**Yes, include the sensitive data in the connection string.**" => click "**Next**" button, choose the radio button "**Entity Framework 6.x**" => click "**Next**" button which will connect to **Data Source** and loads all the objects from there. Under tables select our "**Department**" and "**Employee**" tables and click "**Finish**", which generates all the classes that are necessary for working with tables.

Under solution explorer if we expand “**TestEF.edmx**” item we find “**TestEF.Context.tt**” and if we expand this we find “**TestEF.context.cs**” and if we expand this we find “**MVCDBEntities**” item and if we double click on it, it opens “**TestEF.context.cs**” file and there we find a class with the name “**MVCDBEntities**” (**DataContext** class) inheriting from “**DbContext**”. **DbContext** is the main class that is responsible for interacting with the **Database** and it can perform the following activities:

- **Querying:** Converts “**LINQ-to-Entities**” Queries to **SQL Queries** and sends them to the **Database**.
- **Change Tracking:** Keeps track of changes that occurred on the **entities** after **Querying** from the **Database**.
- **Persisting Data:** Performs the **Insert, Update and Delete** operations to the **Database**, based on entity states.
- **Caching:** Provides first level caching by default. It stores the entities which have been retrieved during the life time of a context class.
- **Manage Relationship:** Manages relationships using CSDL, MSL and SSDL in Db-First or Model-First approach, and using fluent API configurations in Code-First approach.
- **Object Materialization:** Converts raw data from the **Database** into **Entity Objects**.



Under MVCDBEntities class the code will be as following:

```

public partial class MVCDBEntities : DbContext
{
    public MVCDBEntities() : base("name=MVCDBEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Department> Departments { get; set; }
    public virtual DbSet<Employee> Employees { get; set; }
}

```

Note: **Departments** and **Employees** properties of the above class provide a collection of **Department** and **Employee** entities as “**DbSet< TEntity >**” type using which we can access all the data from corresponding tables and perform **CRUD Operations**.

Now expand “**TestEF.tt**” and there we find “**Department.cs**” file which contains the Entity i.e., **Department** class (representing **Department** table) with all of its attributes (**properties** or **columns**), and also “**Employee.cs**” file which contains the Entity i.e., **Employee** class (representing **Employee** table) with all of its attributes (**properties** or **columns**) and in this class we also find a property “**Department**” (**Navigation Property**) which refers to the “**Department**” (1 to 1 relationship) entity which is the parent table for “**Employee**”. Same as the above under “**Department**” class we find a property “**Employees**” (**Navigation Property**) referring to “**Employees**” (1 to many relationship) corresponding to each **Department**.

Under “Department” & “Employee” classes the code will be as following:

```
public partial class Department
{
    public Department()
    {
        this.Employees = new HashSet<Employee>();
    }
    public int Did { get; set; }
    public string Dname { get; set; }
    public string Location { get; set; }
    public virtual ICollection<Employee> Employees { get; set; }
}

public partial class Employee
{
    public int Eid { get; set; }
    public string Ename { get; set; }
    public string Job { get; set; }
    public Nullable<decimal> Salary { get; set; }
    public Nullable<int> Did { get; set; }
    public bool Status { get; set; }
    public virtual Department Department { get; set; }
}
```

Generating Controllers and Views using Scaffolding: we are provided with an option of generating the required **Controllers**, **Action Methods** and **Views** by using **Scaffolding** and to do that first **build** the project, and then right click on the **Controllers** folder and select **“Add” => “Controller”** which opens **“Add New Scaffolded Item”** window, in that window choose **“MVC5 Controller with views, using EntityFramework”** and click on **“Add”** button which opens **“Add Controller”** window , now in that window select **“Model class”** as **“Employee”**, **“Data context class”** as **“MVCDBEntities”**, and in the bottom it will display a name to the controller i.e., **“EmployeesController”** (either leave the same or change to **EmployeeController**) and click on **“Add”** button which will generate a **Controller** with all the required **“Action Methods”** and corresponding views to perform **CRUD Operations**. Run the project and watch the behaviour of the whole application by performing **CRUD Operations**.

Performing CRUD operations manually without using Scaffolding: Open a new **“ASP.NET Web Application”** project naming it as **“MVCWithEFDBF2”**, choose **“MVC Project Template”** and click on **“Create”** button. Now open the solution explorer right click on the Models folder and open the **“Add New Item”** window and select the item **“ADO.NET Entity Data Model”** and name it as **“TestEF.edmx”** and click **“Ok”** which opens a wizard and in that select the option **“EF Designer from database”** and click **“Next”** and in the window opened click on **“New Connection”** button and enter the connection details for our **“MVCDB”** database and then select the radio button **“Yes, include the sensitive data in the connection string.”**, click **“Next”** button, choose the radio button **“Entity Framework 6.x”**, click **“Next”** button which will connect to data sources and load all the object from it, now under tables select our **“Department”** and **“Employee”** tables we have and click **Finish**, which generates all the classes that are necessary for working with database. Now add a controller under Controller’s folder naming it as **“EmployeeController.cs”** and write the below code:

```
using System.Data.Entity;
using MVCWithEFDBF2.Models;
```

```

public class EmployeeController : Controller
{
    MVCDBEntities dc = new MVCDBEntities();
    public ViewResult DisplayEmployees()
    {
        var Emps = dc.Employees.Where(E => E.Status == true);
        return View(Emps);
    }
    public ViewResult DisplayEmployee(int Eid)
    {
        var Emp = dc.Employees.Find(Eid);
        return View(Emp);
    }
    public ViewResult AddEmployee()
    {
        ViewBag.Did = new SelectList(dc.Departments, "Did", "Dname");
        return View();
    }
    [HttpPost]
    public RedirectToRouteResult AddEmployee(Employee Emp)
    {
        Emp.Status = true;
        dc.Employees.Add(Emp);
        dc.SaveChanges();
        return RedirectToAction("DisplayEmployees");
    }
    public ViewResult EditEmployee(int Eid)
    {
        Employee Emp = dc.Employees.Find(Eid);
        ViewBag.Did = new SelectList(dc.Departments, "Did", "Dname", Emp.Did);
        return View(Emp);
    }
    public RedirectToRouteResult UpdateEmployee(Employee Emp)
    {
        Emp.Status = true;
        dc.Entry(Emp).State = EntityState.Modified;
        dc.SaveChanges();
        return RedirectToAction("DisplayEmployees");
    }
    public ViewResult DeleteEmployee(int Eid)
    {
        Employee Emp = dc.Employees.Find(Eid);
        return View(Emp);
    }
    [HttpPost]
    public RedirectToRouteResult DeleteEmployee(Employee Emp)

```

```

{
    //If we want to update the status of employee use the below code:
    dc.Entry(Emp).State = EntityState.Modified;
    //If we want to delete the record permanently comment the above statement and un-comment the below:
    //dc.Entry(Emp).State = EntityState.Deleted;
    dc.SaveChanges();
    return RedirectToAction("DisplayEmployees");
}
}

```

Add a view with the name `DisplayEmployees.cshtml`, selecting layout `Checkbox` and write the below code in it by deleting the whole content in the View:

```

@model IEnumerable<MVCWithEFDBF2.Models.Employee>
 @{
    ViewBag.Title = "Display Employees";
}
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Employee Details</h2>
<table border="1" align="center">
<tr>
    <th>@Html.DisplayNameFor(E => E.Eid)</th>
    <th>@Html.DisplayNameFor(E => E.Ename)</th>
    <th>@Html.DisplayNameFor(E => E.Job)</th>
    <th>@Html.DisplayNameFor(E => E.Salary)</th>
    <th>@Html.DisplayNameFor(E => E.Status)</th>
    <th>@Html.DisplayNameFor(E => E.Did)</th>
    <th>@Html.DisplayNameFor(E => E.Department.Dname)</th>
    <th>@Html.DisplayNameFor(E => E.Department.Location)</th>
    <th>Actions</th>
</tr>
@foreach (var Employee in Model)
{
    <tr>
        <td>@Html.DisplayFor(E => Employee.Eid)</td>
        <td>@Html.DisplayFor(E => Employee.Ename)</td>
        <td>@Html.DisplayFor(E => Employee.Job)</td>
        <td align="right">@Html.DisplayFor(E => Employee.Salary)</td>
        <td align="center">@Html.DisplayFor(E => Employee.Status)</td>
        <td align="center">@Html.DisplayFor(E => Employee.Did)</td>
        <td>@Html.DisplayFor(E => Employee.Department.Dname)</td>
        <td>@Html.DisplayFor(E => Employee.Department.Location)</td>
        <td>
            @Html.ActionLink("View", "DisplayEmployee", new { Eid = Employee.Eid })
            @Html.ActionLink("Edit", "EditEmployee", new { Eid = Employee.Eid })
            @Html.ActionLink("Delete", "DeleteEmployee", new { Eid = Employee.Eid })
        </td>
    </tr>
}

```

```

    }
<tr><td colspan="9" align="center">@Html.ActionLink("Add New Employee", "AddEmployee")</td></tr>
</table>

```

Generate a view for `DisplayEmployee` action method and while adding the View, choose the **Template as Empty, Model Class as Employee**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Employee Details</h2>
<table border="1" align="center">
    <tr><td>Eid:</td><td>@Model.Eid</td></tr>
    <tr><td>Ename:</td><td>@Model.Ename</td></tr>
    <tr><td>Job:</td><td>@Model.Job</td></tr>
    <tr><td>Salary:</td><td>@Model.Salary</td></tr>
    <tr><td>Status:</td><td>@Html.DisplayFor(E => E.Status)</td></tr>
    <tr><td>Did:</td><td>@Model.Did</td></tr>
    <tr><td>Dname:</td><td>@Model.Department.Dname</td></tr>
    <tr><td>Location:</td><td>@Model.Department.Location</td></tr>
    <tr>
        <td colspan="2" align="center">@Html.ActionLink("Back to Employee Details", "DisplayEmployees")</td>
    </tr>
</table>

```

Generate a view for `AddEmployee` action method and while adding the View, choose the **Template as Empty, Model Class as Employee**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Employee</h2>
@using (Html.BeginForm("AddEmployee", "Employee"))
{
    <div>@Html.LabelFor(E => E.Ename)<br />@Html.TextBoxFor(E => E.Ename)</div>
    <div>@Html.LabelFor(E => E.Job)<br />@Html.TextBoxFor(E => E.Job)</div>
    <div>@Html.LabelFor(E => E.Salary)<br />@Html.TextBoxFor(E => E.Salary)</div>
    <div><label>Department</label><br />@Html.DropDownList("Did", "-Select Department-")</div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" name="btnReset" />
    </div>
}
@Html.ActionLink("Back to Employee Details", "DisplayEmployees")

```

Generate a view for `EditEmployee` action method and while adding the View, choose the **Template as Empty, Model Class as Employee**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Employee</h2>
@using (Html.BeginForm("UpdateEmployee", "Employee"))
{
    @Html.HiddenFor(E => E.Eid)

```

```

<div>@Html.LabelFor(E => E.Ename)<br />@Html.TextBoxFor(E => E.Ename)</div>
<div>@Html.LabelFor(E => E.Job)<br />@Html.TextBoxFor(E => E.Job)</div>
<div>@Html.LabelFor(E => E.Salary)<br />@Html.TextBoxFor(E => E.Salary)</div>
<div><label>Department</label><br />@Html.DropDownList("Did")</div>
<div>
    <input type="submit" value="Update" name="btnUpdate" />
    @Html.ActionLink("Cancel", "DisplayEmployees")
</div>
}

```

Generate a view for `DeleteEmployee` action method and while adding the View, choose the **Template as Empty**, **Model Class as Employee**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Delete Employee</h2>
@using (Html.BeginForm("DeleteEmployee", "Employee"))
{
    <table border="1">
        <tr><td>Eid:</td><td>@Html.TextBoxFor(E => E.Eid, new { @readonly = "true" })</td></tr>
        <tr><td>Ename:</td><td>@Html.TextBoxFor(E => E.Ename, new { @readonly = "true" })</td></tr>
        <tr><td>Job:</td><td>@Html.TextBoxFor(E => E.Job, new { @readonly = "true" })</td></tr>
        <tr><td>Salary:</td><td>@Html.TextBoxFor(E => E.Salary, new { @readonly = "true" })</td></tr>
        <tr><td>Did:</td><td>@Html.TextBoxFor(E => E.Did, new { @readonly = "true" })</td></tr>
    </table>
    <font color="red">Are you sure of deleting the current record?</font>
    <input type="submit" value="Yes" /><br />
    @Html.ActionLink("Cancel", "DisplayEmployees")
}

```

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:

```
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:

```
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 = Emps[0].Department;

```

Calling Stored Procedures and performing CRUD Operations

Create a new “**ASP.NET Web Application**” project naming it as “**MVCWithEFDBF3**”, choose “**MVC Project Template**” and click on “**Create**” button. Now open Solution Explorer, right click on the **Models** folder, open the “**Add New Item**” window and select the item “**ADO.NET Entity Data Model**” naming it as “**TestEF.edmx**” and click “**Ok**”, this opens a wizard and in that select the option “**EF Designer from database**” and click “**Next**”, in the window opened click on “**New Connection**” button and enter the connection details for our “**MVCDB**” database and then select the radio button “**Yes, include the sensitive data in the connection string.**” and click “**Next**” button, in the new window choose the radio button “**Entity Framework 6.x**” and click “**Next**” button which will connect to data sources and load all the object from it, now under **Stored Procedures** node select our **Student_Select, Student_Insert, Student_Update** and **Student_Delete** Stored Procedures we have created earlier and click **Finish**, which generates **Methods** mapping with **Stored Procedures** under **MVCDBEntities** class and the **Method** names will be same as **Procedure** names.

In our Procedures, we have a Select **Procedure** which is fetching 5 columns out of the 6 columns present in our table so mapping to the results of this Stored Procedure, it defines a new class whose name will be **<ProcedureName>_Result**, because the procedure name is **Student_Select**, the class name will be **“Student_Select_Result”** and this class will contain 5 properties representing the 5 columns we are fetching. To view that class, open Solution Explorer and under Models folder expand **TestEF.edmx**, under that expand **TestEF.tt**, under that we find a file **Student_Select_Result.cs** which contains the class **Student_Select_Result** and this class is what we will be using for model binding the views, which will be as following:

```

public partial class Student_Select_Result {
    public int Sid { get; set; }
    public string Name { get; set; }
    public Nullable<int> Class { get; set; }
    public Nullable<decimal> Fees { get; set; }
    public string Photo { get; set; }
}

```

Note: even if we are fetching all 6 columns of the table also, still it creates this class and in such cases we can delete this class and map the results with original **Student** class.

Add a folder under the project naming it as “**Uploads**” and then add a new **Controller** class in **Controllers** folder naming it as **StudentController.cs** and write the below code in it:

```

public class StudentController : Controller
{
    MVCDBEntities dc = new MVCDBEntities();
    public ViewResult DisplayStudents()
    {
        return View(dc.Student_Select(null, true));
    }
    public ViewResult DisplayStudent(int sid)
    {
        return View(dc.Student_Select(sid, true).Single());
    }
    public ViewResult AddStudent()
    {
        Student_Select_Result student = new Student_Select_Result();
        return View(student);
    }
    [HttpPost]
    public RedirectToRouteResult AddStudent(Student_Select_Result student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            string PhysicalPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(PhysicalPath))
                Directory.CreateDirectory(PhysicalPath);
            selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
            student.Photo = selectedFile.FileName;
        }
        dc.Student_Insert(student.Sid, student.Name, student.Class, student.Fees, student.Photo);
        return RedirectToAction("DisplayStudents");
    }
    public ViewResult EditStudent(int sid)
    {
        var student = dc.Student_Select(sid, true).Single();
        TempData["Photo"] = student.Photo;
    }
}

```

```

        return View(student);
    }

    public RedirectToRouteResult UpdateStudent(Student_Select_Result student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            string PhysicalPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(PhysicalPath))
                Directory.CreateDirectory(PhysicalPath);
            selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
            student.Photo = selectedFile.FileName;
        }
        else if (TempData["Photo"] != null)
            student.Photo = TempData["Photo"].ToString();
        dc.Student_Update(student.Sid, student.Name, student.Class, student.Fees, student.Photo);
        return RedirectToAction("DisplayStudents");
    }

    public ViewResult DeleteStudent(int sid)
    {
        return View(dc.Student_Select(sid, true).Single());
    }

    [HttpPost]
    public RedirectToRouteResult DeleteStudent(Student_Select_Result student)
    {
        dc.Student_Delete(student.Sid);
        return RedirectToAction("DisplayStudents");
    }
}

```

Add a view with the name **DisplayStudent.cshtml**, selecting layout Checkbox and write the below code in it by deleting the whole content in the View:

```

@model IEnumerable<MVCWithEFDBF3.Models.Student_Select_Result>
 @{
    ViewBag.Title = "Display Students";
}

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center">
    <tr>
        <th>@Html.DisplayNameFor(S => S.Sid)</th>
        <th>@Html.DisplayNameFor(S => S.Name)</th>
        <th>@Html.DisplayNameFor(S => S.Class)</th>
        <th>@Html.DisplayNameFor(S => S.Fees)</th>
        <th>@Html.DisplayNameFor(S => S.Photo)</th>
        <th>Actions</th>
    </tr>
    </tr>
    @foreach (var Student in Model)

```

```

{
    <tr>
        <td>@Html.DisplayFor(S => Student.Sid)</td>
        <td>@Html.DisplayFor(S => Student.Name)</td>
        <td>@Html.DisplayFor(S => Student.Class)</td>
        <td>@Html.DisplayFor(S => Student.Fees)</td>
        <td><img src='/Uploads/@Student.Photo' width='85' height='30' alt='No Image' /></td>
        <td>
            @Html.ActionLink("View", "DisplayStudent", new { Sid = Student.Sid })
            @Html.ActionLink("Edit", "EditStudent", new { Sid = Student.Sid })
            @Html.ActionLink("Delete", "DeleteStudent", new { Sid = Student.Sid })
        </td>
    </tr>
}
<tr><td colspan="6" align="center">@Html.ActionLink("Add New Student", "AddStudent")</td></tr>
</table>

```

Generate a view for **DisplayStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class as Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center">
    <tr>
        <td rowspan="4"><img src='/Uploads/@Model.Photo' width="200" height="200" alt="No Image" /></td>
        <td>Sid: @Model.Sid</td>
    </tr>
    <tr><td>Name: @Model.Name</td></tr>
    <tr><td>Class: @Model.Class</td></tr>
    <tr><td>Fees: @Model.Fees</td></tr>
    <tr><td colspan="2" align="center">@Html.ActionLink("Back to Student Details", "DisplayStudents")</td></tr>
</table>

```

Generate a view for **AddStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class as Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
@using (Html.BeginForm("AddStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid)</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
    <div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
    <div>@Html.LabelFor(S => S.Photo)<br /><input type="file" name="selectedFile" /></div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" />
    </div>
}

```

```

    </div>
    @Html.ActionLink("Back to Student Details", "DisplayStudents")
}

```

Generate a view for **EditStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class as Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Student</h2>
@using (Html.BeginForm("UpdateStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data"}))
{
    <div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid, new { @readonly = true })</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
    <div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
    <div>
        @Html.LabelFor(S => S.Photo)<br />
        <img src='/Uploads/@Model.Photo' width="100" height="100" />
        <input type="file" name="selectedFile" />
    </div>
    <div>
        <input type="submit" value="Update" name="btnUpdate" />
        @Html.ActionLink("Cancel", "DisplayStudents")
    </div>
}

```

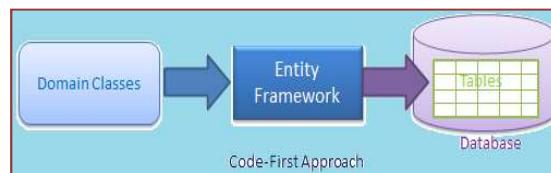
Generate a view for **DeleteStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class as Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Delete Student</h2>
@using (Html.BeginForm("DeleteStudent", "Student")) {
    <table border="1">
        <caption>Student Details</caption>
        <tr>
            <td rowspan="4"><img src='/Uploads/@Model.Photo' width="200" height="200" /></td>
            <td>Sid: @Model.Sid @Html.HiddenFor(S => S.Sid)</td>
        </tr>
        <tr><td>Name: @Model.Name</td></tr>
        <tr><td>Class: @Model.Class</td></tr>
        <tr><td>Fees: @Model.Fees</td></tr>
    </table>
    <font color="red">Are you sure of deleting the current record?</font>
    <input type="submit" value="Yes" />
}
@using (Html.BeginForm("DisplayStudents", "Student")) {
    <text>Click</text> <input type="submit" value="No" /> @:for going back to Student Details.
}
```

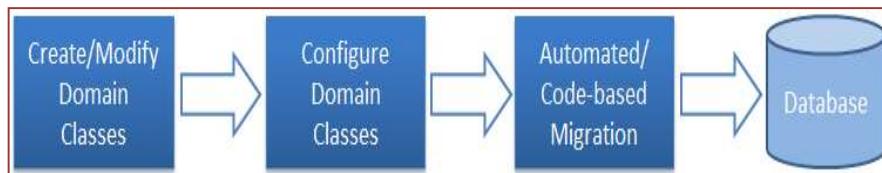
Code-First Approach

Entity Framework introduced the Code-First approach with Entity Framework 4.1. Code-First is mainly useful in Domain Driven Design. In the Code-First approach, we focus on the domain of your application and start creating classes for our domain entity rather than design our database first and then create the classes which match your database design. The following figure illustrates the code-first approach.



As you can see in the above figure, EF API will create the database based on your domain classes and configurations. This means you need to start coding first in C# or VB.NET languages and then EF will create the database from your code.

Code-First Workflow: The below figure illustrates the code-first development workflow:



To work with Code First Approach, create a new “ASP.NET Web Application” project naming it as “MVCWithEFCF1”, choose “MVC Project Template” and click on the “Create” button. Now do the following actions:

Step 1: Install Entity Framework in our project and to do that open “Nuget Package Manager”, go to Browse tab, search for “Entity Framework” and install “Entity Framework by Microsoft”. This will add all the references that are required to work with Entity Framework in our project.

Step 2: Add 3 Classes into the Models folder naming them as “Category.cs”, “Product.cs” and “StoreDbContext.cs”, and we call them as Domain Classes and then write the below code under them:

```
public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public ICollection<Product> Products { get; set; }
}
```

```
public class Product
{
    public int Id { get; set; }
    public string ProductName { get; set; }
    public int CategoryId { get; set; }
    public decimal UnitPrice { get; set; }
}
```

```
public byte[] ProductImage { get; set; }
public string ProductImageName { get; set; }
public bool Discontinued { get; set; }
public Category Category { get; set; }
}
```

```
using System.Data.Entity;
public class StoreDbContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

Step 3: Go to “`Web.config`” file and write the below code inside of `<configuration></configuration>` tag:

```
<connectionStrings>
<add name="StoreDbContext" connectionString="Data Source=Server;User Id=Sa;Password=123;
Database=StoreDB" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Step 4: Go to “`_Layout.cshtml`” and add 2 menus to work with `Categories` and `Products`. To do this, add 2 `ActionLinks` below the following statement:

```
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
```

Write the below code, under the above statement:

```
<li>@Html.ActionLink("Categories", "DisplayCategories", "Category")</li>
<li>@Html.ActionLink("Products", "DisplayProducts", "Product")</li>
```

Step 5: Add a controller in `Controllers` folder naming it as “`CategoryController.cs`”, define all the required `Action` methods and create `Views` that are necessary to perform `CRUD` operations on “`Categories`” table.

```
using System.Data.Entity;
using MVCWithEFCF1.Models;
public class CategoryController : Controller
{
    StoreDbContext dc = new StoreDbContext();
    public ViewResult DisplayCategories()
    {
        var categories = dc.Categories;
        return View(categories);
    }
    public ViewResult AddCategory()
    {
        return View();
    }
}
```

```

[HttpPost]
public RedirectToRouteResult AddCategory(Category category)
{
    dc.Categories.Add(category);
    dc.SaveChanges();
    return RedirectToAction("DisplayCategories");
}
public ViewResult EditCategory(int CategoryId)
{
    Category category = dc.Categories.Find(CategoryId);
    return View(category);
}
public RedirectToRouteResult UpdateCategory(Category category)
{
    dc.Entry(category).State = EntityState.Modified;
    dc.SaveChanges();
    return RedirectToAction("DisplayCategories");
}
public RedirectToRouteResult DeleteCategory(int CategoryId)
{
    Category category = dc.Categories.Find(CategoryId);
    dc.Categories.Remove(category);
    dc.SaveChanges();
    return RedirectToAction("DisplayCategories");
}
}

```

Add a **View** with the name **DisplayCategories.cshtml**, selecting layout **Checkbox** and write the below code in it by deleting the whole content in the **View**:

```

@model IEnumerable<MVCWithEFCF1.Models.Category>
 @{
    ViewBag.Title = "Display Categories";
}
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Categories List</h2>
<table border="1" align="center">
<tr>
<th>@Html.DisplayNameFor(C => C.CategoryId)</th>
<th>@Html.DisplayNameFor(C => C.CategoryName)</th>
<th>@Html.DisplayNameFor(C => C.Description)</th>
<th>Actions</th>
</tr>
@foreach (var Category in Model)
{
    <tr>
        <td>@Html.DisplayFor(C => Category.CategoryId)</td>

```

```

<td>@Html.DisplayFor(C => Category.CategoryName)</td>
<td>@Html.DisplayFor(C => Category.Description)</td>
<td>
    @Html.ActionLink("Edit", "EditCategory", new { CategoryId =Category.CategoryId })
    @Html.ActionLink("Delete", "DeleteCategory", new { CategoryId = Category.CategoryId },
                    new { onclick = "return confirm('Are you sure of deleting the record?')"})
</td>
</tr>
}
<tr><td colspan="4" align="center">@Html.ActionLink("Add New Category", "AddCategory")</td></tr>
</table>

```

Generate a view for **AddCategory** action method and while adding the View, choose the **Template as Empty**, **Model Class as Category**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Category</h2>
@using (Html.BeginForm("AddCategory", "Category"))
{
    <div>@Html.LabelFor(C => C.CategoryName)<br />@Html.TextBoxFor(C => C.CategoryName)</div>
    <div>@Html.LabelFor(C => C.Description)<br />@Html.TextAreaFor(C => C.Description)</div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" name="btnReset" />
    </div>
}
@Html.ActionLink("Back to Category Details", "DisplayCategories")

```

Generate a view for **EditCategory** action method and while adding the View, choose the **Template as Empty**, **Model Class as Category**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Update Category</h2>
@using (Html.BeginForm("UpdateCategory", "Category"))
{
    <div>
        @Html.LabelFor(C => C.CategoryId)<br />@Html.TextBoxFor(C => C.CategoryId, new { @readonly = "true" })
    </div>
    <div>@Html.LabelFor(C => C.CategoryName)<br />@Html.TextBoxFor(C => C.CategoryName)</div>
    <div>@Html.LabelFor(C => C.Description)<br />@Html.TextAreaFor(C => C.Description)</div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        @Html.ActionLink("Cancel", "DisplayCategories")
    </div>
}

```

Step 6: Add another controller in **Controllers** folder naming it as “**ProductController.cs**”, define all the required **Action** methods and also create **Views** that are necessary to perform **CRUD** operations on “**Products**” table.

```
using System.IO;
using System.Data.Entity;
using MVCWithEFCF1.Models;
public class ProductController : Controller
{
    StoreDbContext dc = new StoreDbContext();
    public ViewResult DisplayProducts()
    {
        dc.Configuration.LazyLoadingEnabled = false;
        var products = dc.Products.Include(P => P.Category).Where(P => P.Discontinued == false);
        return View(products);
    }
    public ViewResult DisplayProduct(int Id)
    {
        dc.Configuration.LazyLoadingEnabled = false;
        Product product = (dc.Products.Include(P => P.Category).Where(
            P => P.Id == Id && P.Discontinued == false)).Single();
        return View(product);
    }
    public ViewResult AddProduct()
    {
        ViewBag.CategoryId = new SelectList(dc.Categories, "CategoryId", "CategoryName");
        return View();
    }
    [HttpPost]
    public RedirectToRouteResult AddProduct(Product product, HttpPostedFileBase selectedFile)
    {
        if(selectedFile != null)
        {
            string DirectoryPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(DirectoryPath))
            {
                Directory.CreateDirectory(DirectoryPath);
            }
            selectedFile.SaveAs(DirectoryPath + selectedFile.FileName);
            BinaryReader br = new BinaryReader(selectedFile.InputStream);
            product.ProductImage = br.ReadBytes(selectedFile.ContentLength);
            product.ProductImageName = selectedFile.FileName;
        }
        dc.Products.Add(product);
        dc.SaveChanges();
        return RedirectToAction("DisplayProducts");
    }
}
```

```

public ViewResult EditProduct(int Id)
{
    Product product = dc.Products.Find(Id);
    TempData["ProductImage"] = product.ProductImage;
    TempData["ProductImageName"] = product.ProductImageName;
    ViewBag.CategoryId = new SelectList(dc.Categories, "CategoryId", "CategoryName", product.CategoryId);
    return View(product);
}
public RedirectToRouteResult UpdateProduct(Product product, HttpPostedFileBase selectedFile)
{
    if (selectedFile != null)
    {
        string DirectoryPath = Server.MapPath("~/Uploads/");
        if (!Directory.Exists(DirectoryPath))
        {
            Directory.CreateDirectory(DirectoryPath);
        }
        selectedFile.SaveAs(DirectoryPath + selectedFile.FileName);
        BinaryReader br = new BinaryReader(selectedFile.InputStream);
        product.ProductImage = br.ReadBytes(selectedFile.ContentLength);
        product.ProductImageName = selectedFile.FileName;
    }
    else if(TempData["ProductImage"] != null && TempData["ProductImageName"] != null) {
        product.ProductImage = (byte[])TempData["ProductImage"];
        product.ProductImageName = (string)TempData["ProductImageName"];
    }
    dc.Entry(product).State = EntityState.Modified;
    dc.SaveChanges();
    return RedirectToAction("DisplayProducts");
}
public RedirectToRouteResult DeleteProduct(int Id)
{
    Product product = dc.Products.Find(Id);
    product.Discontinued = true;
    dc.Entry(product).State = EntityState.Modified;
    dc.SaveChanges();
    return RedirectToAction("DisplayProducts");
}
}

```

Add a **View** with the name **DisplayProducts.cshtml**, selecting layout **Checkbox** and write the below code in it by deleting the whole content in the **View**:

```

@model IEnumerable <MVCWithEFCF1.Models.Product>
 @{
    ViewBag.Title = "Display Products";

```

```

}

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Display Products</h2>
<table border="1" align="center">
<tr>
<th>@Html.DisplayNameFor(P => P.Id)</th>
<th>@Html.DisplayNameFor(P => P.ProductName)</th>
<th>@Html.DisplayNameFor(P => P.UnitPrice)</th>
<th>@Html.DisplayNameFor(P => P.ProductImage)</th>
<th>@Html.DisplayNameFor(P => P.CategoryId)</th>
<th>@Html.DisplayNameFor(P => P.Category.CategoryName)</th>
<th>@Html.DisplayNameFor(P => P.Category.Description)</th>
<th>Actions</th>
</tr>
@foreach (var Product in Model)
{
<tr>
<td>@Html.DisplayFor(P => Product.Id)</td>
<td>@Html.DisplayFor(P => Product.ProductName)</td>
<td>@Html.DisplayFor(P => Product.UnitPrice)</td>
<td><img src='~/Uploads/@Product.ProductImageName' width="40" height="25" alt="No Image" /></td>
<td>@Html.DisplayFor(P => Product.CategoryId)</td>
<td>@Html.DisplayFor(P => Product.Category.CategoryName)</td>
<td>@Html.DisplayFor(P => Product.Category.Description)</td>
<td>
    @Html.ActionLink("View", "DisplayProduct", new { Id = Product.Id })
    @Html.ActionLink("Edit", "EditProduct", new { Id = Product.Id })
    @Html.ActionLink("Delete", "DeleteProduct", new { Id = Product.Id },
        new { onclick = "return confirm('Are you sure of deleting the record?') } )</td>
</tr>
}
<tr><td colspan="9" align="center">@Html.ActionLink("Add New Product", "AddProduct")</td></tr>
</table>

```

Generate a view for **DisplayProduct** action method and while adding the View, choose the **Template** as **Empty**, **Model Class** as **Product**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Display Product</h2>
<table border="1" align="center">
<tr>
<td rowspan=7>
    <img src='~/Uploads/@Model.ProductImageName' width="200" height="200" alt="No Image" />
</td>
<td>ProductId: @Model.Id</td>
</tr>
<tr><td>ProductName: @Model.ProductName</td></tr>

```

```

<tr><td>UnitPrice: @Model.UnitPrice</td></tr>
<tr><td>CategoryId: @Model.CategoryId</td></tr>
<tr><td>CategoryName: @Model.Category.CategoryName</td></tr>
<tr><td>Description: @Model.Category.Description</td></tr>
<tr><td colspan="2" align="center">@Html.ActionLink("Back to Product Details", "DisplayProducts")</td></tr>
</table>

```

Generate a view for `AddProduct` action method and while adding the View, choose the **Template as Empty**, **Model Class as Product**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “`<h2>`” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Product</h2>
@using (Html.BeginForm("AddProduct", "Product", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(P => P.ProductName)<br />@Html.TextBoxFor(P => P.ProductName)</div>
    <div>@Html.LabelFor(P => P.UnitPrice)<br />@Html.TextBoxFor(P => P.UnitPrice)</div>
    <div>@Html.LabelFor(P => P.ProductImage)<br /><input type="file" name="selectedFile" /></div>
    <div>@Html.Label("Category Name")<br />@Html.DropDownList("CategoryId", "-Select Category-")</div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" />
    </div>
}
@Html.ActionLink("Back to Product Details", "DisplayProducts")

```

Generate a view for `EditProduct` action method and while adding the View, choose the **Template as Empty**, **Model Class as Product**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “`<h2>`” element:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Update Product</h2>
@using (Html.BeginForm("UpdateProduct", "Product", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(P => P.Id)<br />@Html.TextBoxFor(P => P.Id, new { @readonly = "true" })</div>
    <div>@Html.LabelFor(P => P.ProductName)<br />@Html.TextBoxFor(P => P.ProductName)</div>
    <div>@Html.LabelFor(P => P.UnitPrice)<br />@Html.TextBoxFor(P => P.UnitPrice)</div>
    <div>
        @Html.LabelFor(P => P.ProductImage)<br />
        <img src='/Uploads/@Model.ProductImageName' width="100" height="100" /><br />
        <input type="file" name="selectedFile" />
    </div>
    <div>@Html.LabelFor(P => P.CategoryId)<br />@Html.DropDownList("CategoryId")</div>
    <div>
        <input type="submit" value="Update" name="btnUpdate" />
        @Html.ActionLink("Cancel", "DisplayProducts")
    </div>
}

```

Step 7: Now if we want the **Database** and **Tables** to be created on the **Database Server**, we need to perform **Migrations**. **Migration** is a feature “**Entity Framework Code First**” provides which is used to keep the “**Domain Classes**” and “**Database**” in **sync** with each other.

Now when we run the application it launches the “**HomeController’s**”, “**Index View**” and the **Menu** in **Layout** will display “**Categories**” and “**Products**” links on top of the **Page** and when we click on the “**Categories**” link it will immediately create the **Database** on the server with the name “**StoreDB**”.

To test this, run the project, click on “**Categories**” link on the top and then it will display the View “**Category List**” providing options for adding a new category. Now go to **SQL Server Database** and watch, we will find a new **Database** with the name “**StoreDB**” (we specified this in **Web.config** file) and under the **Database** we will find **Categories** and **Products** tables (names of tables are taken based on **DbSet** properties in **DbContext** class).

Note: in our “**Web.config**” file we specified the name of “**Connection String**” as “**StoreDbContext**” and this name is the same name of our “**DbContext**” class, so we don’t require writing any code to read the “**ConnectionString**” into our application. If we want to give our own name for **Connection String** in “**Web.config**” without using the name “**StoreDbContext**”, we need to explicitly specify that name by defining a default constructor in “**StoreDbContext**” class and call its **base** class constructor using “**base**” keyword and pass the “**Connection String**” name as a parameter to it. For example, in “**Web.config**”, if we give the name of “**Connection String**” as “**ConStr**” but not “**StoreDbContext**”, then we need to define a constructor in our **Context** class as below:

```
public StoreDbContext() : base("ConStr")
{
}
```

Database Initialization Strategies: right now, when we run the project for first time it will create the “**Database and Tables**” and from next time on wards it will use the same “**Database and Tables**” when we run the application. There are various **options** to decide for us whether we want to use the existing **Database** next time or **re-create** it every time, and that is based on **Database Initialization Strategies**. To handle this, we have to use one of the **Database Initialization Strategies** which should be specified in our **Context** class constructor as following:

```
Database.SetInitializer(new CreateDatabaseIfNotExists<ContextClassName>());
```

Entity Framework Supports 3 different initialization strategies like:

CreateDatabaseIfNotExists: This is the **default initializer**. As the name suggests, it will create the **Database** if none exists as per the configuration. However, if you change the model class and then run the application with this initializer, then it will throw an **Exception**.

DropCreateDatabaseIfModelChanges: This initializer drops an existing **Database** and creates a new if your **model** classes (entity classes) have been changed. So, you don’t have to worry about maintaining your **Database** schema when your model classes change.

DropCreateDatabaseAlways: As the name suggests, this initializer drops an existing **Database** every time you run the application, irrespective of whether your **model** classes have changed or not.

To try them go to constructor of our Context class i.e., “StoreDbContext**” and use any one of the following:**

```
Database.SetInitializer(new CreateDatabaseIfNotExists<StoreDbContext>()); //Default Strategy
Database.SetInitializer(new DropCreateDatabaseIfModelChanges<StoreDbContext>());
Database.SetInitializer(new DropCreateDatabaseAlways<StoreDbContext>());
```

Turn off the DB Initializer: you can even turn off the **Database initializer** for your application if you don't want to lose existing data in **production environment**, then you can turn off **initializer**, as shown below:

```
Database.SetInitializer<StoreDbContext>(null);
```

EF 6 Code-First Conventions: conventions are a set of default rules which automatically configure a conceptual model based on our domain classes when working with the **Code-First** approach. As you have seen in the previous example, **EF API** configured **Primary Keys**, **Foreign Keys**, **Relationships**, and **Column Data Types** etc. from the domain classes without any additional configurations. This is because of the **EF Code-First conventions**. If they are followed in domain classes, then the **Database** schema will be configured based on the conventions. These **EF 6.x Code-First conventions** are defined in the **"System.Data.Entity.ModelConfiguration.Conventions"** namespace.

The following table lists default code first conventions

Default Convention For	Description
Schema	By default, EF creates all the Database objects into the dbo schema.
Table Name	<Entity Class Name> + 's' EF will create a Database table with the entity class name suffixed by ' s ' for example "Student" domain class (entity) would map to "Students" table.
Primary Key Name	1) A property with the name "Id" . 2) <Entity Class Name> + "Id" (case in-sensitive) EF will create a primary key column for the property named Id or <Entity Class Name> + "Id" (case in-sensitive).
Foreign Key Property Name	By default, EF will look for the foreign key property with the same name as the principal entity primary key name. If the foreign key property does not exist, then EF creates an FK Column in the table with <Dependent Navigation Property Name> + _ + <Principal Entity Primary Key Property Name>, for example EF will create "Category_CategoryId" foreign key column in Products table if the Product entity does not contain a foreign key property.
Null Column	EF creates a null column for all reference type properties and nullable value properties e.g., string, Nullable<int> or int?, etc.
Not Null Column	EF creates Not Null columns for Primary Key properties and non-nullable value type properties e.g., int, float, bool, decimal, DateTime etc.
DB Columns Order	EF will create DB columns in the same order as the properties in an entity class. However, primary key columns would be moved first.
Properties Mapping to DB	By default, all properties will map to the Database. Use the [NotMapped] attribute to exclude property or class from DB mapping.
Cascade Delete & Update	Enabled by default for all types of relationships.

The following table list C# data types mapped with SQL Server data types

C# Data Type	Mapping to SQL Server Data Type
int	Int
string	nvarchar(Max)
decimal	decimal(18,2)
float	Real
byte[]	varbinary(Max)
datetime	Datetime
bool	Bit
byte	Tinyint
short	Smallint
long	Bigint
double	float
char	No Mapping (Throws Exception)
sbyte	No Mapping (Throws Exception)
object	No Mapping (Throws Exception)

Data Annotations: Data Annotations are .NET attributes which can be applied on an entity class or properties to override default conventions in EF 6 and EF Core. Data Annotations are provided in EF 6 and EF Core under the namespaces `System.ComponentModel.DataAnnotations` and `System.ComponentModel.DataAnnotations.Schema`. These attributes can be used in Entity Framework as well as with ASP.NET MVC Data Controls.

System.ComponentModel.DataAnnotations Attributes:

Attribute	Description
<code>Key</code>	Can be applied to a property to specify a key property in an entity and make the corresponding column a “Primary Key” column in the Database.
<code>Timestamp</code>	Can be applied to a property to specify the data type of a corresponding column in the Database as “Row Version”.
<code>Required</code>	Can be applied to a property to specify that the corresponding column is a “Not Null” column in the Database.
<code>MinLength</code>	Can be applied to a property to specify the minimum string length allowed in the corresponding column in the Database.
<code>MaxLength</code>	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the Database.
<code>StringLength</code>	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the Database.

System.ComponentModel.DataAnnotations.Schema Attributes:

Attribute	Description
<u>Table</u>	Can be applied to an entity class to configure the corresponding table name and schema in the Database.
<u>Column</u>	Can be applied to a property to configure the corresponding column name, order and data type in the Database.
<u>Index</u>	Can be applied to a property to configure that the corresponding column should have an Index in the Database. (EF 6.1 onwards only)
<u>ForeignKey</u>	Can be applied to a property to mark it as a foreign key property.
<u>NotMapped</u>	Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database.
<u>DatabaseGenerated</u>	Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g., identity, computed or none.
<u>InverseProperty</u>	Can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship.
<u>ComplexType</u>	Marks the class as complex type in EF 6. EF Core 2.0 does not support this attribute.

To test overriding the default Code First Conventions with Data Annotations create a new “[ASP.Net Web Application](#)” project naming it as “[MVCWithEFCF2](#)”, choose MVC Project Template and click on “[Create](#)” button. Install Entity Framework in the project and then add 3 new classes into the [Models](#) folder naming them as “[Supplier.cs](#)”, “[Customer.cs](#)” and “[CompanyDbContext.cs](#)”, and write the below code under them:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
[Table("Supplier")] //Specifying Name for the table being created in database.
public class Supplier
{
    [Key] //Setting this column as Primary Key column.
    [DatabaseGenerated(DatabaseGeneratedOption.None)] //Setting identity off, so will not autogenerate values.
    public int Sid { get; set; }

    [MaxLength(100)] //Setting MaxLength as 100 for this column.
    [Column("Sname", TypeName = "Varchar")] //Setting column name and data type of the column.
    public string SupplierName { get; set; }

    public ICollection<Customer> Customers { get; set; }
}

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
[Table("Customer")]
public class Customer
{
```

```

[Column(TypeName = "Money")]
public decimal? Balance { get; set; }

[Index] //Setting Index attribute for this column in database.
[Required] //Setting not null constraint for this column in database.
[MaxLength(50)]
[Column("Cname", TypeName = "Varchar")]
public string CustomerName { get; set; }

[StringLength(1000)]
[Column(TypeName = "Varchar")]
public string Address { get; set; }

[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public int Custid { get; set; }

public int SupplierId { get; set; }

[ForeignKey("SupplierId")] //Setting the foreign key column for the table.
public Supplier Supplier { get; set; }
}

```

```

using System.Data.Entity;
public class CompanyDbContext : DbContext
{
    public CompanyDbContext() : base("ConStr")
    {
        Database.SetInitializer(new DropCreateDatabaseAlways<CompanyDbContext>());
    }
    public DbSet<Supplier> Suppliers { get; set; }
    public DbSet<Customer> Customers { get; set; }
}

```

Now open Web.config file and write the following code between <configuration></configuration> tag:

```

<connectionStrings>
<add name="ConStr"
      connectionString="Data Source=Server;Database=CompanyDB;User Id=Sa;Password=123"
      providerName="System.Data.SqlClient" />
</connectionStrings>

```

Add a controller naming it as SupplierController and write the below code under it by deleting existing code:

```

using MVCWithEFCF2.Models;
public class SupplierController : Controller
{
    CompanyDbContext dc = new CompanyDbContext();

```

```

public ActionResult Index()
{
    Supplier s1 = new Supplier { Sid = 101, SupplierName = "Ashok Distributors." };
    Supplier s2 = new Supplier { Sid = 102, SupplierName = "Meghna Distributors." };
    Supplier s3 = new Supplier { Sid = 103, SupplierName = "Diamond Distributors." };
    Supplier s4 = new Supplier { Sid = 104, SupplierName = "Prasad Distributors." };
    dc.Suppliers.Add(s1); dc.Suppliers.Add(s2);
    dc.Suppliers.Add(s3); dc.Suppliers.Add(s4);
    dc.SaveChanges();
    return View(dc.Suppliers);
}
}

```

Add a view to the `Index` action method and while adding the View, in “Add View” window choose “Template:” as “List”, “Model class:” as “Supplier (`MVCWithEFCF2.Models`)”, “Data context class:” as “`CompanyDbContext (MVCWithEFCF2.Models)`” and click on “Add” button. Run the view we have created which will create the Database and Tables under SQL Server.

Seed Data in Entity Framework Code-First: We can insert data into our Database tables during the Database initialization process. This will be important when we want to provide some “default - master data” for any table in the application, for example in our previous application if we want to insert default data into “Department” table then we need to use the concept of “Seed” and insert default data into the table. To seed data into any table, we need to implement the logic in “Seed” method of any 3 DBInitializer classes (“`CreateDatabaseIfNotExists`”, “`DropCreateDatabaseAlways`”, “`DropCreateDatabaseIfModelChanges`”) and to do that we have to define a custom “`DBInitializer`” class inheriting from any of the above 3 “`DBInitializer`” classes and then override the “`Seed`” method. To test this process, add a new class in `Models` folder of our previous project with the name “`CompanyDBInitializer.cs`” and write the below code in it:

```

using System.Data.Entity;
public class CompanyDBInitializer : DropCreateDatabaseIfModelChanges<CompanyDbContext>
{
    protected override void Seed(CompanyDbContext context)
    {
        Supplier s1 = new Supplier { Sid = 101, SupplierName = "Ashok Distributors." };
        Supplier s2 = new Supplier { Sid = 102, SupplierName = "Meghna Distributors." };
        Supplier s3 = new Supplier { Sid = 103, SupplierName = "Diamond Distributors." };
        Supplier s4 = new Supplier { Sid = 104, SupplierName = "Prasad Distributors." };
        context.Suppliers.Add(s1); context.Suppliers.Add(s2);
        context.Suppliers.Add(s3); context.Suppliers.Add(s4);
        context.SaveChanges();
    }
}

```

Go to “`CompanyDbContext`” class and re-write the constructor of the class as below:

```

public CompanyDbContext() : base("ConStr")
{
}

```

```
        Database.SetInitializer(new CompanyDBInitializer());
    }
```

Go to “SupplierController” class and **delete** the below code, because we have implemented the **Seed** method in “CompanyDBInitializer” class and it will insert data into **Supplier** table during the **Database** initialization process, so this code is no more required in the “SupplierController” class.

```
Supplier s1 = new Supplier { Sid = 101, SupplierName = "Ashok Distributors." };
Supplier s2 = new Supplier { Sid = 102, SupplierName = "Meghna Distributors." };
Supplier s3 = new Supplier { Sid = 103, SupplierName = "Diamond Distributors." };
Supplier s4 = new Supplier { Sid = 104, SupplierName = "Prasad Distributors." };
dc.Suppliers.Add(s1); dc.Suppliers.Add(s2);
dc.Suppliers.Add(s3); dc.Suppliers.Add(s4);
```

Entity Framework Code First Migrations: Entity Framework Code-First has different **Database** initialization strategies like “**CreateDatabaseIfNotExists**”, “**DropCreateDatabaseIfModelChanges**”, “**DropCreateDatabaseAlways**” however, there are **problems** with these strategies. In the first case **Model** changes will not be updated to the **Database** and also throws an error when we run the project, whereas in the second and third cases **Model** changes will be updated to the **Database** but if we already have data (other than seed data) or we created our own **Stored Procedures**, **Triggers**, **Views** etc. in our **Database** then these strategies will drop the entire **Database** and recreates it, so we will lose the **Data** and **Database Objects** also.

To overcome the above problems “**Entity Framework**” introduced “**Migrations**” that automatically updates the **Database** schema whenever our **model** changes without losing any existing **Data** or other **Database Objects**. To do that we need to use a new **Database Initializer** called “**MigrateDatabaseToLatestVersion**”.

There are 2 kinds of Migration available like:

- Automated Migration
- Code-based Migration

Automated Migration: Entity Framework provides **automated migration** option so that you don't have to process **Database** migration **manually** for each change you make in your domain classes. The **automated migrations** can be implemented by executing the “**enable-migrations**” command in the **Package Manager Console**.

To test “**Automated Migrations**”, create a new “**ASP.NET Web Application**” project naming it as “**MVCWithEFCF3**”, choose “**MVC Project Template**” and click on “**Create**” button. Install **Entity Framework**, add 2 classes under **Models** folder with names “**Student**” and “**SchoolDbContext**” and write the below code under them:

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
}

using System.Data.Entity;
public class SchoolDbContext : DbContext
{
```

```

public SchoolDbContext() : base("ConStr")
{
}
public DbSet<Student> Students { get; set; }
}

```

Go to Web.config file and write the below code between <configuration></configuration> tag:

```

<connectionStrings>
<add name="ConStr" providerName="System.Data.SqlClient"
      connectionString="Data Source=Server;Database=SchoolDB;User Id=Sa;Password=123" />
</connectionStrings>

```

Add a controller under the Controllers folder naming it as “SchoolController” and write the below code in it:

```

using MVCWithEFCF3.Models;
public class StudentController : Controller
{
    SchoolDbContext dc = new SchoolDbContext();
    public ActionResult Index()
    {
        Student s = new Student { Name = "Raju" };
        dc.Students.Add(s);
        dc.SaveChanges();
        return View();
    }
}

```

Add a view to “Index” action method; run the project which will create “SchoolDB” Database under SQL Server and in that “Student” Table is created with 2 Columns and 1 record is inserted into the Table.

Now if we make any changes to the Model class like adding new attributes (**properties**) or modifying existing attributes and run the project we get an error because by default the **Database Initialization Strategy** is “CreateDatabaseIfNotExists” and if we try to change the **Database Initialization Strategy** to “DropCreateDatabaseIfModelChanges” or “DropCreateDatabaseAlways” we will be losing the existing data in the table and to overcome this problem let’s use “**Migrations**”.

Automatic Migrations: to use Automatic Migrations we need to enable migrations in our project and to do that go to “**Package Manager Console**” window from “Tools Menu” => “Nuget Package Manager” => “**Package Manager Console**” and run the below command there:

PM> Enable-Migrations -EnableAutomaticMigration:\$true

The above command will add a “**Migrations**” folder in our project and under that folder we find a file “**Configuration.cs**” which contains a class called “**Configuration**” with the below code in it:

```

internal sealed class Configuration : DbMigrationsConfiguration<MVCWithEFCF3.Models.SchoolDbContext>
{
    public Configuration()
    {
}

```

```

{
    AutomaticMigrationsEnabled = true;
    ContextKey = "MVCWithEFCF3.Models.SchoolDbContext";
}
protected override void Seed(MVCWithEFCF3.Models.SchoolDbContext context) {
    //This method will be called after migrating to the latest version.
    //You can use the DbSet<T>.AddOrUpdate() helper extension method to avoid creating duplicate seed data.
}
}

```

Now go to our “**SchoolDbContext**” class and write the below statement in its constructor by importing the namespace “**MVCWithEFCF3.Migrations**”.

```
Database.SetInitializer(new MigrateDatabaseToLatestVersion<SchoolDbContext, Configuration>());
```

With this action everything is set ready for **Automatic Migrations**, and to test this either add new properties to **Student** class or add new **Model** classes in the project and the next time when we run the project, we see all the changes reflecting under the **Database**. However, sometimes when you modify or remove existing properties in domain classes we get “**System.Data.Entity.Migrations.Infrastructure.AutomaticDataLossException**”, and this is because we may lose data in corresponding column of the **Table**. So, to handle this kind of scenario, we have to set “**AutomaticMigrationDataLossAllowed = true**” in the “**Configuration**” class **Constructor**. To do that go to “**Configuration**” class in “**Migrations**” folder and add the above statement which should now look as following:

```

public Configuration()
{
    AutomaticMigrationsEnabled = true;
    AutomaticMigrationDataLossAllowed = true;           //New Statement to be added
    ContextKey = "MVCWithEFCF3.Models.SchoolDbContext";
}

```

Code Based Migrations: code-based migration provides more control on the **migration** and allows you to configure additional things such as setting a default value to a column, **change data type of a column**, **add not null constraints on a column**, **configure a computed column** etc.

To use code-based migration we need to execute the following commands in “**Package Manager Console**” window under **Visual Studio**:

- **Enable-Migrations:** enables the migration in your project by creating a Configuration class.
- **Add-Migration:** creates a new migration class as per specified name with the Up() and Down() methods.
- **Update-Database:** executes the last migration file created by the Add-Migration command and applies changes to the database schema.

To test **Code Based Migrations** create a new “**ASP.NET Web Application**” project naming it as “**MVCWithEFCF4**”, choose “**MVC Project Template**” and click on “**Create**” button. Install **Entity Framework** in the project, add 2 new classes in to **Models** folder naming them as “**Student**” and “**SchoolDbContext**”, and write the below code under them:

```

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Class { get; set; }

    [MaxLength(1)]
    [Column(TypeName = "Varchar")]
    public string Section { get; set; }
}

using System.Data.Entity;
public class SchoolDbContext : DbContext
{
    public SchoolDbContext() : base("ConStr")
    {
    }
    public DbSet<Student> Students { get; set; }
}

```

Go to Web.config file and write the below code between <configuration></configuration> tag:

```

<connectionStrings>
    <add name="ConStr" providerName="System.Data.SqlClient"
        connectionString="Data Source=Server;Database=SchoolDB;User Id=Sa;Password=123" />
</connectionStrings>

```

Add a controller under the Controllers folder naming it as “SchoolController” and write the below code in it:

```

using MVCWithEFCF4.Models;
public class StudentController : Controller
{
    SchoolDbContext dc = new SchoolDbContext();
    public ActionResult Index()
    {
        Student s1 = new Student { Name = "Raju", Class = 10, Section = "A" };
        Student s2 = new Student { Name = "Venkat", Class = 10, Section = "B" };
        Student s3 = new Student { Name = "Srinivas", Class = 10, Section = "C" };
        dc.Students.Add(s1); dc.Students.Add(s2); dc.Students.Add(s3);
        dc.SaveChanges();
        return View();
    }
}

```

Add a **View** to **Index** action method and run the project which will create **“SchoolDB”** Database on **SQL Server** and under that **Database, “Student”** Table is created with **4 columns** in it.

Working with Code Based Migrations: To work with code-based migrations, first execute the “enable-migrations” command in the “Package Manager Console” and to do that go to Tools menu => Nuget Package Manager => Package Manager Console and run the below command over there:

PM>Enable-Migrations

This command will create the Configuration class same as we saw in case of “Automated Migrations” but in this case Configuration class Constructor will have “AutomaticMigrationsEnabled = false” because we are using Code Based Migrations now, and this command also creates a “<timestamp>_InitialCreate.cs” file containing a class “InitialCreate” with “Up()” and “Down()” methods as following:

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable("dbo.Students", c => new
        {
            StudentId = c.Int(nullable: false, identity: true),
            Name = c.String(),
            Class = c.Int(nullable: false),
            Section = c.String(maxLength: 1, unicode: false),
        }).PrimaryKey(t => t.StudentId);
    }

    public override void Down()
    {
        DropTable("dbo.Students");
    }
}
```

As we can see the “Up()” method contains code for creating Database objects and the “Down()” method contains code for dropping or deleting Database objects. You may also write your own custom code for additional configurations and that is the advantage of Code Based Migrations over Automated Migrations.

Now go to “SchoolDbContext” class and write the below code in its constructor by importing the namespace “MVCWithEFC4.Migrations”:

```
Database.SetInitializer(new MigrateDatabaseToLatestVersion<SchoolDbContext, Configuration>());
```

From now whenever we make changes to existing Model classes or add new Model classes, we need to create a new Migration class by using the “Add-Migration” command in the “Package Manager Console” with a name to the class (any name) as following:

Syntax => PM> Add-Migration <Some Name>

To test this, go to Student class and add a new property into the class as following:

```
public float Fees { get; set; }
```

Open Package Manager Console (PMC) and create a new migration file by executing the following statement:

PM> Add-Migration SchoolDB-V1

The above action will create a new migration file with the name “<timestamp>_SchoolDB-v1.cs” and in the same way we can create any no. of migration files as above whenever we make changes to the **Model** classes and to discriminate between each file give a different name every time, for example:

```
PM>Add-Migration SchoolDB-v2  
PM>Add-Migration SchoolDB-v3  
PM>Add-Migration SchoolDB-v4
```

After creating a migration file as above, we must execute the “**Update-Database**” command in the “**Package Manager Console**” and update the **Database** as below:

```
PM>Update-Database
```

Note: to view the **SQL** statements being applied to the target **Database** use “**-Verbose**” option as below:

```
PM>Update-Database -Verbose
```

The above statement when executed will run the last or latest migration file and alters the **Database** based on the changes, we have made to domain classes.

Let's now add another 2 properties in the “**Student**” class and to do that go to “**Student.cs**” file and write the below code in the class:

```
public float Marks { get; set; }  
public string Address { get; set; }
```

Create another migration file at the Package Manager Console as following:

```
PM> Add-Migration SchoolDB-V2
```

Note: the above action will create a new migration file with the name “<timestamp>_SchoolDB-v2.cs”.

Get Migrations: right now, we have 3 migrations files that are created but what we have applied on the **Database** is only 2 and to check what migration are applied on the **Database** use the “**Get-Migrations**” command at **Package Manager Console** as below:

```
PM>Get-Migrations
```

Now let's apply the 3rd migration file also to the **Database** i.e. “**SchoolDB-V2**” and to do that run the following statement at **Package Manager Console**:

```
PM>Update-Database -Verbose
```

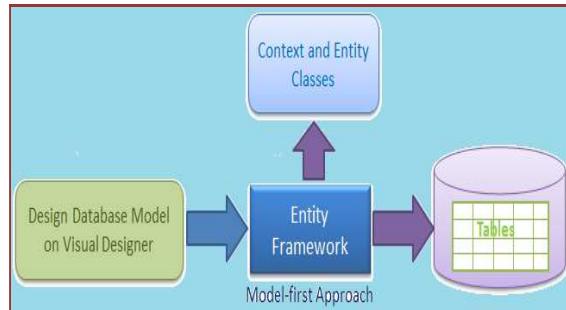
Rollback Migration: suppose if we want to roll back the **Database** schema to any of the previous states, then you can execute the “**update-database**” command with the “**-TargetMigration**” parameter to the point which you want to roll back to. For example, suppose there are many migrations applied to the above “**SchoolDB**” Database, but you want to roll back to “**SchoolDB-V1**” migration then execute the below command:

```
PM>Update-Database -TargetMigration:SchoolDB-V1
```

Note: this statement will execute the “**Down**” method in the migration file and drops the 2 columns from the **Database** table, which are added in “**SchoolDB-V2**” migration. This action will not delete the migration file under “**Migrations**” folder, so if we don't want it, we need to explicitly delete the file on our own.

Model-First Approach

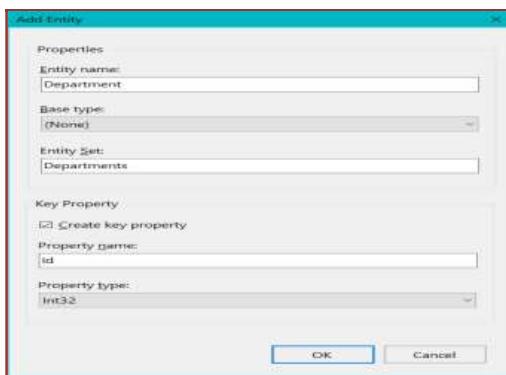
In **Model-First** approach, we create **entities**, **relationships**, and **inheritance hierarchies** directly on **Visual Designer** integrated in **Visual Studio**, which will then generate **Model Classes**, **Context Class**, and also the **Database Script** from your **Visual Model**.



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

To work with **Entity Framework Model First Approach**, create a new “**ASP.NET Web Application**” project naming it as “**MVCWithEPMF**”, choose “**MVC Project Template**” and click on “**Create**” button. Open **Solution Explorer**, right click on the **Models** folder and choose “**Add => New Item**” option and in the window opened select the item “**ADO.NET Entity Data Model**” name it as “**TestEF**” and click on the “**Add**” button and in the window opened select “**Empty EF Designer Model**” and click on “**Finish**” button. This will add “**TestEF.edmx**” item under **Models** folder and under this item we will find 2 items with the names “**TestEF.Designer.cs**” and “**TestEF.edmx.diagram**”. We also see “**TestEF.edmx [Diagram1]**” displayed in document window and by using this we can design **Model** classes.

Creating a Department Entity: now right click on the “**TestEF.edmx [Diagram1]**”, in the document window and select the option “**Add New** => “**Entity**”, which will open “**Add Entity**” window => enter the “**Entity name**” as “**Department**” and this will automatically fill the other details in the window, watch them and click on “**Ok**” button which will add the “**Department**” entity with “**Id**” attribute.



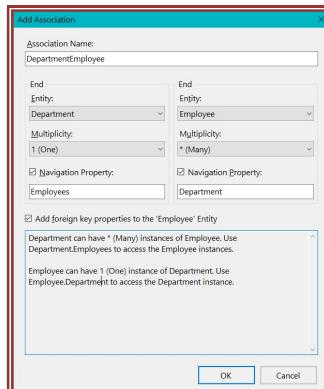
Adding new Properties to Department Entity: right click on the **Department Entity** and select “**Add New** => “**Scalar Property**” which will add a new property name it as “**Dname**”, right click on “**Dname**” property and select the option “**Properties**” which opens “**Property Window**” and in that window we can set various attributes like **Type**, **Nullable**, **Length** etc, and by default the **Type** will be “**String**” which is ok for “**Dname**” so leave it and set the “**Max Length**” property value as “**50**”, “**Unicode**” property value as “**false**” and “**Nullable**” property value as “**false**”. Repeat the same process again and add another property with the name “**Location**” and set the “**Max Length**” property value as “**50**”, “**Unicode**” property value as “**false**” and “**Nullable**” property value as “**true**”.

Creating an Employee Entity: right click on the document window and select the option “Add New” => “Entity”, which will open “Add Entity” window => enter the “Entity name” as “Employee” which will automatically fill the other details, watch them, and click on the “Ok” button which will add the “Employee” entity with “Id” attribute.

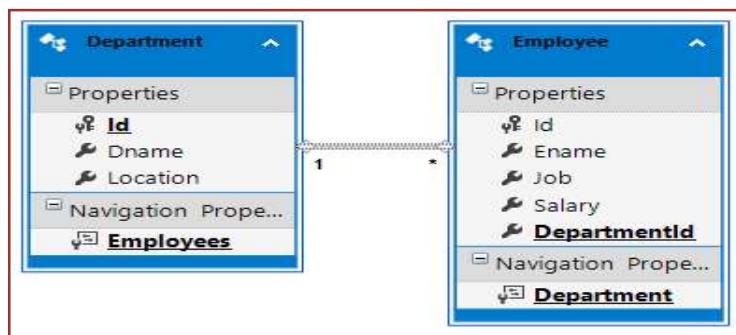
Adding new Properties to Employee Entity: now add all the other required properties to “Employee” Entity with the following names and attribute values:

Ename: Type: String, Max Length: 50, Unicode: False, Nullable: False
Job: Type: String, Max Length: 50, Unicode: False, Nullable: True
Salary: Type: Decimal, Precision: 9, Scale: 2, Nullable: True
Status: Type: Boolean, Default Value: True, Nullable: False

Adding a relationship between the tables: to add a relationship between the 2 tables right click, on the document window and select the option “Add New” => “Association” and in the window opened it will display the following details, verify them, change if anything is not as per our requirements (as of now everything is perfectly set, so no need to make any changes) and click on the “Ok” button.



This will add a “DepartmentId” column to the “Employee” entity, also adds navigation properties in both the entities and this will also create a relationship between the tables as following:



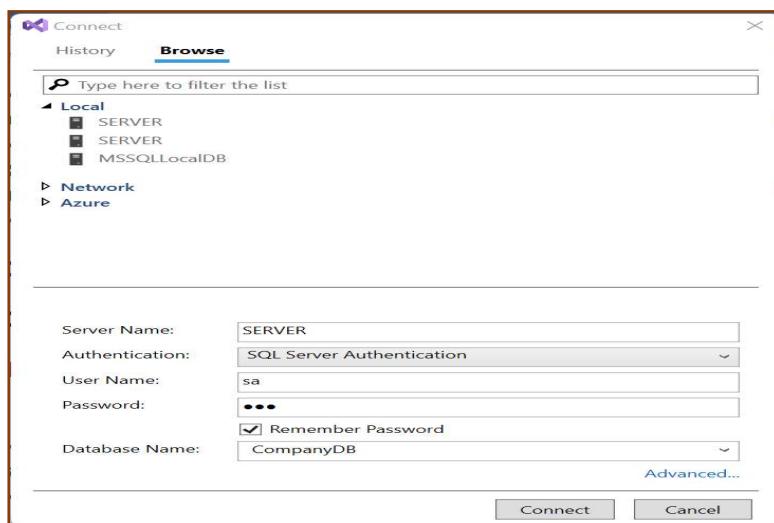
Generating Model Classes and Database with Tables, using the Visual Model: right click on the document window and select the option “Generate Database from Model” which will open a window and in that click on “New Connection” button and enter the connection details like “Server Name”, “Authentication Details” and in “Select or enter a database name.” TextBox enter Database Name as “CompanyDB” (make sure this Database is not existing under SQL Server) and when we click on the “Ok” button it will ask for creating the Database because it is not existing,

click on the “Yes” button which will create the **Database** and launches a new window, here select the radio button “Yes, include the sensitive data in the connection string.” => Click “Next” and select the Entity Framework version i.e., “6.x” => Click “Next” which displays the required “SQL Script” code that is generated for creating **Tables** and this code will be saved into our “**Models**” folder with the name “TestEF.edmx.sql” and click on the “Finish” button to complete the configuration which will also install “Entity Framework” in our project.

Now open “**Solution Explorer**” and watch the “TestEF.edmx” item under “**Models**” folder there we find new items added with the names “TestEF.Context.tt” and “TestEF.tt”. We find a “**Context**” class with the name “TestEFContainer” under the file “TestEF.Context.cs” which is present under “TestEF.Context.tt” item, and we also find **Model** classes under “TestEF.tt” item, which when expanded will show “Department.cs” and “Employee.cs” files containing “**Department**” and “**Employee**” classes.

Note: In the Context class i.e., “**TestEFContainer**” Constructor, we find a call to its base or parent class **Constructor** and under that we find the name of “**Connection String**” which is nothing but “**TestEFContainer**”, so in the “**Web.config**” file we will find the “**Connection String**” with this name. Even if it is not required to specify the name and call base class constructor because we have already learnt that if the “**Context**” class name and “**Connection String**” name are same no need to specify that explicitly, but “**Entity Framework Model First**” will include that code in **Constructor**.

Creating Tables on Database Server: after all the above actions i.e., **Entity Framework** generating required **Context** and **Model Classes**, now we need to create the tables under our **Database** i.e., “**CompanyDB**” and to create the tables, inside of the **Model’s** folder we find a **SQL Script** file with name “TestEF.edmx.sql” which is generated by “**Entity Framework Code First**” which we need to execute and to do that, open the script file, right click on it in document window and select the option “**Execute**” which will open “**Connect**” window, in that expand the node “**Local**” and under that we find our “**Server Name**” (Server in my case) select it, and in the below specify the **Authentication Details** and choose the **Database** as “**CompanyDB**” under “**Database Name**” DropDownList and click connect which will create the tables on **Database**.



Note: Now we can start working with those **Tables** and **Domain Classes** by creating the required **Controllers**, **Action Methods**, and **Views**.

ADO.NET

Pretty much every application deal with data in some manner, whether that data comes from memory, databases, XML files, text files, or something else. The location where we store the data can be called as a Data Source or Data Store where a Data Source can be a file, database, address books or indexing server etc.

Programming Languages cannot communicate with Data Sources directly because each Data Source adopts a different Protocol (set of rules) for communication, so to overcome this problem long back Microsoft has introduced intermediate technologies like ODBC and Ole DB which works like bridge between the Applications and Data Sources to communicate with each other.

ODBC (Open Database Connectivity) is a standard C programming language middleware API for accessing database management systems (DBMS). ODBC accomplishes DBMS independence by using an ODBC driver as a translation layer between the application and the DBMS. The application uses ODBC functions through an ODBC driver manager with which it is linked, and the driver passes the query to the DBMS. An ODBC driver will be providing a standard set of functions for the application to use and implementing DBMS-specific functionality. An application that can use ODBC is referred to as "ODBC-Compliant". Any ODBC-Compliant application can access any DBMS for which a driver is installed. Drivers exist for all major DBMS's as well as for many other data sources like Microsoft Excel, and even for Text or CSV files. ODBC was originally developed by Microsoft in 1992.

- It's a collection of drivers, where these drivers sit between the App's and Data Source's to communicate with each other and more over we require a separate driver for each data source.
- ODBC drivers comes along with your Windows O.S. and we can find them at the following location:

Control Panel => Administrative Tools => ODBC Data Sources

- To consume these ODBC Drivers first we need to configure them with the data source by creating a "DSN" (Data Source Name).
- ODBC drivers are open source i.e., there is an availability of these ODBC Drivers for all the leading O.S's in the market.

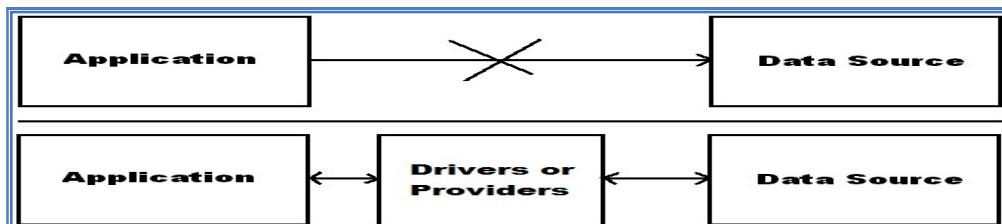
Drawbacks with ODBC Drivers:

These drivers must be installed on every machine where the application is executing from and then the application, driver and data source should be manually configured with each other.

ODBC Drivers are initially designed for communication with Relational DB only.

OLE DB (Object Linking and Embedding, Database, sometimes written as OLEDB or OLE-DB), an API designed by Microsoft, allows accessing data from a variety of data sources in a uniform manner. The API provides a set of interfaces implemented using the Component Object Model (COM) and SQL. Microsoft originally intended OLE DB as a higher-level replacement for, and successor to, ODBC, extending its feature set to support a wider variety of non-relational databases, such as object databases and spreadsheets that do not necessarily implement SQL. OLE DB is conceptually divided into consumers and providers. The consumers are the applications that need access to the data, and the providers are the software components that implement the interface and thereby provide the data to the consumer. An OLE DB provider is a software component enabling an OLE DB consumer to interact with a data source. OLE DB providers are alike to ODBC drivers. OLE DB providers can be created to access such simple data stores as a text file and spreadsheet, through to such complex databases as Oracle, Microsoft SQL Server, and many others. It can also provide access to hierarchical data stores. These OLE DB Providers are introduced by Microsoft around the year 1996.

- It's a collection of providers where these providers sit between the App's and Data Source to communicate with each other, and we require a separate provider for each data source.
- OLE DB Providers are designed for communication with relational & non-relational Data Sources also i.e. it provides support for communication with any Data Source.
- OLE DB Providers sits on server machine so they are already configured with data source and when we connect with any data source they will help in the process of communication.
- OLE DB Providers are developed by using COM and SQL Languages, so they are also un-managed.
- Microsoft introduced OLEDB as a replacement for ODBC for its Windows Systems.
 - OLE DB is a pure Microsoft technology which works only on Windows Platform.



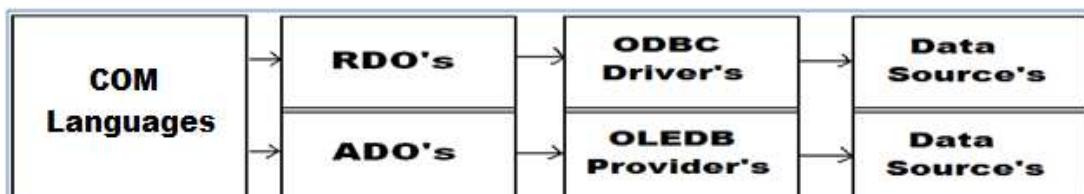
Things to remember while working with ODBC and OLE DB:

- ODBC and OLE DB are un-managed or platform dependent.
- ODBC and OLE DB are not designed targeting any language i.e., they can be consumed by any language like: C, CPP, Visual Basic, Visual CPP, Java, CSharp etc.

Note: If any language wants to consume ODBC Drivers or OLE DB Providers, they must use some built-in libraries of the language in which we are developing the application without writing complex coding.

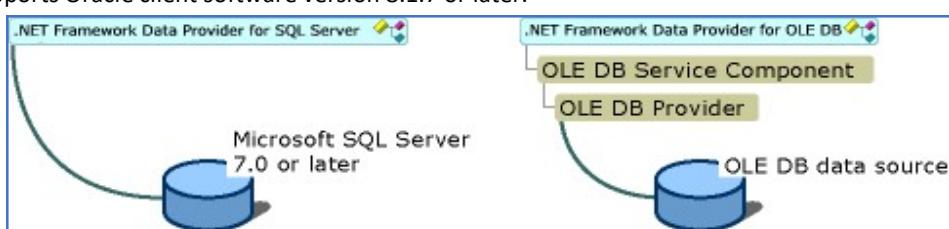
RDO's and ADO's in COM Language:

COM Language used RDO's (Remote Data Objects) and ADO's (ActiveX Data Objects) for data source communication without having to deal with the comparatively complex ODBC or OLEDB API.



.NET Framework Providers:

The .NET Framework Data Provider for SQL Server uses its own protocol to communicate with SQL Server. It is lightweight and performs well because it is optimized to access a SQL Server directly without adding an OLE DB or ODBC layer and it supports SQL Server software version 7.0 or later. The .NET Framework Data Provider for Oracle (Oracle Client) enables data access to Oracle data sources through Oracle client connectivity software. The data provider supports Oracle client software version 8.1.7 or later.



ADO.NET: It is a set of types that expose data access services to the .NET programmer. ADO.NET provides functionality to developers writing managed code like the functionality provided to native COM developers by ADO. ADO.NET provides consistent access to data sources such as Microsoft SQL Server, as well as data sources exposed through OLE DB and XML. Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, manipulate, and update data. It is an integral part of the .NET Framework, providing access to relational data, XML, and application data. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects used by applications or Internet browsers.

ADO.Net provides libraries for Data Source communication under the following namespaces:

- System.Data
- System.Data.Odbc
- System.Data.OleDb
- System.Data.SqlClient
- System.Data.OracleClient

Note: System.Data, System.Data.Odbc, System.Data.OleDb and System.Data.SqlClient namespaces are under the assembly System.Data.dll whereas System.Data.OracleClient is under System.Data.OracleClient.dll assembly.

System.Data: types of this namespace are used for holding and managing of data on client machines. This namespace contains following set of classes in it: **DataSet**, **DataTable**, **DataRow**, **DataColumn**, **DataView**, **DataRelation**, etc.

System.Data.Odbc: types of this namespace can communicate with any Relational Data Source using Un-Managed Odbc Drivers.

System.Data.OleDb: types of this namespace can communicate with any Data Source using OleDb Providers (Un-Managed COM Providers).

System.Data.SqlClient: types of this namespace can purely communicate with Sql Server database only using SqlCommand Provider (Managed .Net Framework Provider).

System.Data.OracleClient: types of this namespace can purely communicate with Oracle database only using OracleClient Provider (Managed .Net Framework Provider).

All the above 4 namespaces contain same set of types as following: **Connection**, **Command**, **DataReader**, **DataAdapter**, **Parameter** and **CommandBuilder** etc, but here each class is referred by prefixing with Odbc, OleDb, Sql and Oracle keywords before the class name to discriminate between each other as following:

OdbcConnection	OdbcCommand	OdbcDataReader	OdbcDataAdapter	OdbcCommandBuilder	OdbcParameter
OleDbConnection	OleDbCommand	OleDbDataReader	OleDbDataAdapter	OleDbCommandBuilder	OleDbParameter
SqlConnection	SqlCommand	SqlDataReader	SqlDataAdapter	SqlCommandBuilder	SqlParameter
OracleConnection	OracleCommand	OracleDataReader	OracleDataAdapter	OracleCommandBuilder	OracleParameter

Performing Operations on a DataSource: the operations we perform on a Data Source will be Select, Insert, Update and Delete, and each operation we perform on a Data Source involves in 3 steps, like:

- Establishing a connection with the data source.
- Sending a request to Data Source by using SQL.
- Capturing the results given by the data source.

Establishing a Connection with Data Source: It's a process of opening a channel for communication between Application and Data Source that is present either on a local or remote machine to perform DB Operations and to open the channel for communication we use Connection class.

Constructors of the Class:

Connection()

Connection(string ConnectionString)

Note: ConnectionString is a collection of attributes that are required for connecting with a DataSource, those are:

- DSN
- Provider
- Data Source
- User Id and Password
- Integrated Security
- Database or Initial Catalog
- Extended Properties

DSN: this is the only attribute that is required if we want to connect with a data source by using ODBC Drivers and by using this attribute, we need to specify the DSN Name.

Provider: this attribute is required when we want to connect to the data source by using Oledb Providers. So, by using this attribute we need to specify the provider's name based on the data source we want to connect with.

Oracle:	Msdaora or ORAOLEDB.ORACLE	Sql Server:	SqlOledb
MS-Access or MS-Excel:	Microsoft.Jet.Oledb.4.0	MS-Indexing Server:	Msidxs

Data Source: this attribute is required to specify the server's name if the data source is a database or else if the data source is a file, we need to specify path of the file and this attribute is required in case of any provider communication.

User Id and Password: This attribute is required to specify the credentials for connection with a database and this attribute is required in case of any provider communication.

Oracle: Scott/tiger	SQL Server: Sa/123
---------------------	--------------------

Integrated Security: this attribute is used while connecting with [SQL Server Database only](#) to specify that we want to connect with the Server by using Windows Authentication and in this case, we should not use User Id and Password attributes and this attribute is required in case of any provider communication.

Database or Initial Catalog: these attributes are used while connecting with [SQL Server Database only](#) to specify the name of DB we want to connect with, and this attribute is required in case of any provider communication.

Extended Properties: this attribute is required only while connecting with MS-Excel using Oledb Provider.

List of attributes which are required in case of Odbc Drivers, OleDb and .Net Framework Providers:

<u>Attribute</u>	<u>ODBC Driver</u>	<u>OLEDB Provider</u>	<u>.NET Framework Provider</u>
DSN	Yes	No	No
Provider	No	Yes	No
Data Source	No	Yes	Yes
User Id and Password	No	Yes	Yes
Integrated Security*	No	Yes	Yes
Database or Initial Catalog*	No	Yes	Yes
Extended Properties**	No	Yes	-

*Only for Sql Server

**Only for Microsoft Excel

Connection String for SqlServer to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");  
OleDbConnection con = new OleDbConnection("Provider=SqlOleDb;Data Source=<Server Name>;  
Database=<DB Name>;User Id=<User Name>;Password=<Pwd>");  
SqlConnection con = new SqlConnection("Data Source=<Server Name>;Database=<DB Name>;  
User Id=<User Name>;Password=<Pwd>");
```

Note: in case of Windows Authentication in place of User Id and Password attributes we need to use Integrated Security = SSPI (Security Support Provider Interface).

Connection String for Oracle to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");  
OleDbConnection con = new OleDbConnection("Provider=Msaora (o)r ORAOLEDB.ORACLE;  
Data Source=<Server Name>;User Id=<User Name>;Password=<Pwd>");  
OracleConnection con = new OracleConnection("Data Source=<Server Name>;  
User Id=<User Name>;Password=<Pwd>");
```

Connection String for MS-Excel to connect by using different options:

```
OdbcConnection con = new OdbcConnection("Dsn=<Dsn Name>");  
OleDbConnection con = new OleDbConnection("Provider=Microsoft.Jet.Oledb.4.0;  
Data Source=<Path of Excel Document>;Extended Properties=Excel 8.0");
```

Members of Connection class:

Open(): a method which opens a connection with data source.

Close(): a method which closes the connection that is open.

State: an enumerated property which is used to get the status of connection.

ConnectionString: a property which is used to get or set connection string that is associated with connection object.

Object of class Connection can be created in any of the following ways:

```
Connection con = new Connection();  
con.ConnectionString = "<connection string>";  
Or  
Connection con = new Connection("<connection string>");
```

Sending request to Data Source by using SQL: In this process we send a request to Data Source by specifying the type of action we want to perform by using a SQL Statement like Select, Insert, Update, and Delete or by calling a Stored Procedure present under the Data Source. To send and execute SQL Statements or call Stored Procedures in Data Source we use Command class.

Constructors of the class:

```
Command()  
Command(string CommandText, Connection con)
```

Note: CommandText means it can be any SQL Statement like Select or Insert or Update or Delete Statements or Stored Procedure Name, whereas “Connection” refers to instance of Connection class created in 1st step.

Properties of Command Class:

Connection: sets or gets the connection object associated with command object.
CommandText: sets or gets the SQL statement or SP name associated with command object.
CommandType: sets or gets whether Command is configured to execute a SQL Statement [default] or S.P.

The object of class Command can be created in any of the following ways:

```
Command cmd = new Command(); cmd.Connection = <con>; cmd.CommandText = "<Sql Stmt or SP Name>";  
or  
Command cmd = new Command("<Sql Stmt or SP Name>", <con>);
```

Methods of Command class:

ExecuteReader()	=>	DataReader
ExecuteScalar()	=>	object
ExecuteNonQuery()	=>	int

Note: after creating object of Command class, we need to call any of the execute methods to execute the stmt's.

Use **ExecuteReader()** method when we want to execute a Select Statement that returns data as rows and columns. The method returns an object of class DataReader which holds data that is retrieved from data source in the form of rows and columns.

Use **ExecuteScalar()** method when we want to execute a Select Statement that returns a single value result. The method returns result of the query in the form of an object.

Use **ExecuteNonQuery()** method when we want to execute any SQL statement other than select, like Insert or Update or Delete etc. The method returns an integer that tells the no. of rows affected by the statement.

Note: The above process of calling a suitable method to capture results is our third step i.e., capturing the results.

DataReader: it's a class designed for holding the data on client machines in the form of Rows and Columns.

Features of DataReader:

- Can hold multiple tables in it at a time and to load multiple tables into a DataReader pass multiple Select Statements as “CommandText” to command separated by a semi-colon.
- Faster access to data from the Data Source because it is “**Connection Oriented**”.

Drawbacks of DataReader:

- As it is connection oriented requires a continuous connection with data source while we are accessing the data, so there are chances of performance degradation if there are more no. of clients accessing data at the same time.
 - It gives forward only access to the data i.e., allows going either to next record or table but not to previous record or table.
 - It is a read only object which will not allow any changes to data that is present in it.
-

Accessing data from a DataReader: DataReader is a class which can hold data in the form of rows and columns, to access data from DataReader it provides the following members:

1. **GetName(int ColumnIndex)** => **string**

Returns name of the column for given index position.

2. **Read()** => **bool**

Moves record pointer from current location to next row and returns a Boolean value which tells whether the row to where it moved contains any data or not, which will be true if data is present or false if data is not present.

3. **GetValue(int ColumnIndex)** => **object**

4. **Indexer[int ColumnIndex]** => **object**

5. **Indexer[string ColumnName]** => **object**

All the 3 are used for retrieving column values from the row to which pointer was pointing by specifying the Column Index or Column Name.

6. **FieldCount** => **int**

This property returns the no. of columns fetched into the DataReader

7. **NextResult()** => **bool**

Moves record pointer from current table to next table and returns a Boolean value which tells whether the location to which it moved contains a table or not, which will be true if present or false if not present.

Dis-Connected Architecture: ADO.Net supports 2 different models for accessing data from Data Sources:

- Connection Oriented Architecture
- Disconnected Architecture

In the first case we require a continuous connection with Data Source for accessing data from it and in this case, we use DataReader class for holding data on client machines, whereas in the 2nd case we don't require a continuous connection with Data Source for accessing the data from it i.e., we require a connection only for loading data from Data Source and in this case, we use DataSet class for holding data on client machines.

Working with DataSet

DataSet: It's a class present under System.Data namespace designed for holding and managing of the data on client machines apart from DataReader. DataSet class provides the following features:

- DataSet is also capable of holding multiple tables like a DataReader whereas in case of DataSet those tables can be loaded from different Data Sources.
- It is designed in disconnected architecture which requires a connection just for loading data but not for holding and accessing the data.
- It provides scrollable navigation to data which allows us to move in any direction i.e., either top to bottom or bottom to top.
- It is updatable i.e.; changes can be made to data present in it and those changes can be sent back to DB for update.
- It provides options for searching and sorting of data that is present under it.

- It provides options for establishing relations between the tables that are present under it.

Loading Data into DataSet's: The class which is responsible for loading data into DataReader from a DataSource is Command, in the same way DataAdapter class is required for communication between DataSource and DataSet.

DataSource <= Command => DataReader
DataSource <=> DataAdapter <=> DataSet

Note: DataAdapter is internally a collection of 4 commands like “SelectCommand”, “InsertCommand”, “UpdateCommand” and “DeleteCommand” where each command is an instance of Command class, and by using these commands DataAdapter will perform Select, Insert, Update and Delete operations on a table.

Constructors of DataAdapter class:

```
DataAdapter()
DataAdapter(Command SelectCmd)
DataAdapter(string SelectCommandText, Connection con)
DataAdapter(string SelectCommandText, string ConnectionString)
```

Note: Select Command Text means it can be a Select Stmt or a Stored Procedure which contains a Select Stmt.

Instance of DataAdapter class can be created in any of the following ways:

```
Connection con = new Connection("<Connection String>");
Command cmd = new Command("<Select Stmt or SP Name>", con);
DataAdapter da = new DataAdapter();
da.SelectCommand = cmd;
Or
Connection con = new Connection("<Connection String>");
Command cmd = new Command("<Select Stmt or SP Name>", con);
DataAdapter da = new DataAdapter(cmd);
Or
Connection con = new Connection("<Connection String>");
DataAdapter da = new DataAdapter("<Select Stmt or SPName>", con);
or
DataAdapter da = new DataAdapter("<Select Stmt or SPName>", "<Connection String>");
```

Properties of DataAdapter:

- SelectCommand
- InsertCommand
- UpdateCommand
- DeleteCommand

Methods of DataAdapter:

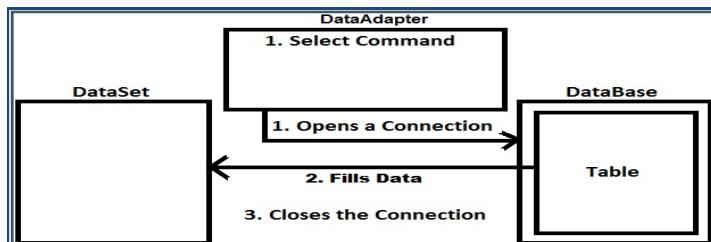
- Fill(DataSet ds, string tableName)
- Update(DataSet ds, string tableName)

Fill method is used for loading data from DataSource into the DataSet and Update method is used for updating any changes made in the DataSet back to DataSource:

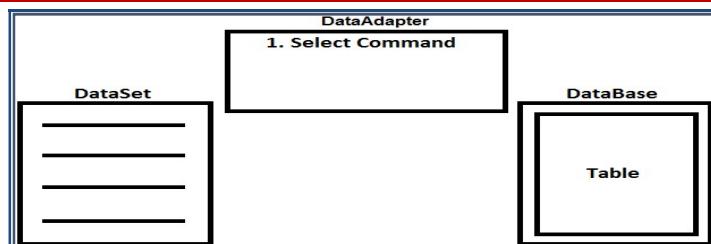
Fill: DataSource => DataAdapter => DataSet
Update: DataSource <= DataAdapter <= DataSet

When we call Fill method on DataAdapter class, then following actions will take place internally:

1. DataAdapter will open a connection with the Data Source.
2. Executes the SelectCommand present in it on the DataSource and loads data from table to DataSet.
3. Closes the connection.

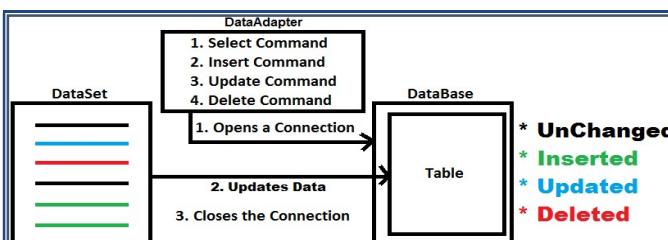


Once the execution of Fill method is completed data gets loaded into the DataSet as below:

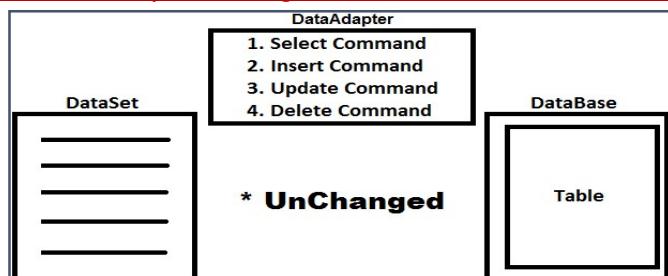


As we are discussing DataSet is updatable, we can make changes to the data that is loaded into it like adding, and modifying and deleting of records and after making changes to data in DataSet if we want to send those changes back to DataSource we need to call Update method on DataAdapter, which performs the following:

1. DataAdapter will re-open the connection with Data Source.
2. Changes that are made to data in DataSet will be sent back to corresponding table, where in this process it will make use of Insert, Update and Delete commands of DataAdapter.
3. Closes the connection.



Once Update method execution is completed data gets re-loaded into DataSet as below with unchanged rows:



Accessing data from DataSet: DataReader's provides pointer-based access to the data, so we can get data only in a sequential order whereas DataSet provides index-based access to the data, so we can get data from any location randomly. DataSet is a collection of tables where each table is represented as a class DataTable and identified by its index position or name. Every DataTable is again collection of Rows and collection of Columns where each row is represented as a class DataRow and identified by its index position and each column is represented as a class DataColumn and identified by its index position or name.

Accessing a DataTable from DataSet: <dataset>.Tables[index] or <dataset>.Tables[name]
E.g.: ds.Tables[0] or ds.Tables["Employee"]

Accessing a DataRow from DataTable: <datatable>.Rows[index]
E.g.: ds.Tables[0].Rows[0]

Accessing a DataColumn from DataTable: <datatable>.Columns[index] or <datatable>.Columns[name]
E.g.: ds.Tables[0].Columns[0] or ds.Tables[0].Columns["Eno"]

Accessing a Cell from DataTable: <datatable>.Rows[row][col]
E.g.: ds.Tables[0].Rows[0][0] or ds.Tables[0].Rows[0]["Eno"]

Create a new **ASP.NET Web Application** project naming it as "**MVCWithADO**", choose "**MVC Project Template**" and click on the "**Create**" button. Under the Model's folder add a class with the name "**Student.cs**" to represent our **Student Entity** and write the below code in it:

```
using System.ComponentModel.DataAnnotations;
public class Student
{
    [Display(Name = "Student Id")]
    public int Sid { get; set; }
    public string Name { get; set; }
    public int? Class { get; set; }
    public decimal? Fees { get; set; }
    public string Photo { get; set; }
}
```

Open Web.config file and add Connection String for connecting to DB under <configuration> tag as following:

```
<connectionStrings>
<add name="ConStr" connectionString="Data Source=Server;Database=MVCDB;User Id=Sa; Password=123"
      providerName="System.Data.SqlClient" />
</connectionStrings>
```

Now add another class in Model's folder with the name "StudentDAL.cs" and write the following code in it:

```
using System.Data;
using System.Configuration;
using System.Data.SqlClient;
public class StudentDAL
{
```

```

SqlCommand cmd;
SqlConnection con;
public StudentDAL()
{
    string ConStr = ConfigurationManager.ConnectionStrings["ConStr"].ConnectionString;
    con = new SqlConnection(ConStr);
    cmd = new SqlCommand();
    cmd.Connection = con;
    cmd.CommandType = CommandType.StoredProcedure;
}
public List<Student> SelectStudent(int? Sid, bool? Status)
{
    List<Student> students = new List<Student>();
    try
    {
        cmd.Parameters.Clear();
        cmd.CommandText = "Student_Select";
        if (Sid != null && Status != null)
        {
            cmd.Parameters.AddWithValue("@Sid", Sid);
            cmd.Parameters.AddWithValue("@Status", Status);
        }
        else if (Sid != null && Status == null)
            cmd.Parameters.AddWithValue("@Sid", Sid);
        else if (Sid == null && Status != null)
            cmd.Parameters.AddWithValue("@Status", Status);
        con.Open();
        SqlDataReader dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            Student student = new Student {
                Sid = (int)dr["Sid"],
                Name = (string)dr["Name"],
                Class = (int)dr["Class"],
                Fees = (decimal)dr["Fees"],
                Photo = (string)dr["Photo"]
            };
            students.Add(student);
        }
    }
    catch(Exception ex)
    { throw ex; }
    finally
    { con.Close(); }
    return students;
}

```

```

public int InsertStudent(Student student)
{
    int Count = 0;
    try
    {
        cmd.CommandText = "Student_Insert";
        cmd.Parameters.Clear();
        cmd.Parameters.AddWithValue("@Sid", student.Sid);
        cmd.Parameters.AddWithValue("@Name", student.Name);
        cmd.Parameters.AddWithValue("@Class", student.Class);
        cmd.Parameters.AddWithValue("@Fees", student.Fees);
        if (student.Photo != null && student.Photo.Length != 0)
            cmd.Parameters.AddWithValue("@Photo", student.Photo);
        con.Open();
        Count = cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    { throw ex; }
    finally
    { con.Close(); }
    return Count;
}

public int UpdateStudent(Student student)
{
    int Count = 0;
    try
    {
        cmd.CommandText = "Student_Update";
        cmd.Parameters.Clear();
        cmd.Parameters.AddWithValue("@Sid", student.Sid);
        cmd.Parameters.AddWithValue("@Name", student.Name);
        cmd.Parameters.AddWithValue("@Class", student.Class);
        cmd.Parameters.AddWithValue("@Fees", student.Fees);
        if (student.Photo != null && student.Photo.Length != 0)
            cmd.Parameters.AddWithValue("@Photo", student.Photo);
        con.Open();
        Count = cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    { throw ex; }
    finally
    { con.Close(); }
    return Count;
}

public int DeleteStudent(int Sid)
{

```

```

int Count = 0;
try
{
    cmd.CommandText = "Student_Delete";
    cmd.Parameters.Clear();
    cmd.Parameters.AddWithValue("@Sid", Sid);
    con.Open();
    Count = cmd.ExecuteNonQuery();
}
catch (Exception ex)
{ throw ex; }
finally
{ con.Close(); }
return Count;
}
}

```

Add a Controller under Controllers folder with the name “StudentController” and write the following code in it:

```

using System.IO;
using MVCWithADO.Models;
public class StudentController : Controller
{
    StudentDAL obj = new StudentDAL();
    public ViewResult DisplayStudents()
    {
        return View(obj.SelectStudent(null, true));
    }
    public ViewResult DisplayStudent(int Sid)
    {
        return View(obj.SelectStudent(Sid, true)[0]);
    }
    public ViewResult AddStudent()
    {
        return View();
    }
    [HttpPost]
    public RedirectToRouteResult AddStudent(Student student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            string PhysicalPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(PhysicalPath))
            {
                Directory.CreateDirectory(PhysicalPath);
            }
            selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
            student.Photo = selectedFile.FileName;
        }
    }
}

```

```

        }

        obj.InsertStudent(student);
        return RedirectToAction("DisplayStudents");
    }

    public ViewResult EditStudent(int Sid)
    {
        Student student = obj.GetStudents(Sid, true).Single();
        TempData["Photo"] = student.Photo;
        return View(s);
    }

    public RedirectToRouteResult UpdateStudent(Student student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            string PhysicalPath = Server.MapPath("~/Uploads/");
            if (!Directory.Exists(PhysicalPath))
            {
                Directory.CreateDirectory(PhysicalPath);
            }
            selectedFile.SaveAs(PhysicalPath + selectedFile.FileName);
            student.Photo = selectedFile.FileName;
        }
        else
        {
            student.Photo = TempData["Photo"].ToString();
        }
        obj.UpdateStudent(student);
        return RedirectToAction("DisplayStudents");
    }

    public RedirectToRouteResult DeleteStudent(int Sid)
    {
        obj.DeleteStudent(Sid);
        return RedirectToAction("DisplayStudents");
    }
}

```

Add a view with the name `DisplayStudents.cshtml`, selecting layout `Checkbox` and write the below code in it by deleting the whole content in the `View`:

```

@model IEnumerable<MVCWithADO.Models.Student>
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center" class="table-condensed">
<tr>
<th>@Html.DisplayNameFor(S => S.Sid)</th>
<th>@Html.DisplayNameFor(S => S.Name)</th>
<th>@Html.DisplayNameFor(S => S.Class)</th>
<th>@Html.DisplayNameFor(S => S.Fees)</th>

```

```

<th>@Html.DisplayNameFor(S => S.Photo)</th>
<th align="center">Actions</th>
</tr>
@foreach (MVCWithADO.Models.Student student in Model)
{
<tr>
<td>@Html.DisplayFor(S => student.Sid)</td>
<td>@Html.DisplayFor(S => student.Name)</td>
<td>@Html.DisplayFor(S => student.Class)</td>
<td>@Html.DisplayFor(S => student.Fees)</td>
<td><img src='/Uploads/@student.Photo' width="40" height="25" alt="No Image" />
</td>
<td>
    @Html.ActionLink("View", "DisplayStudent", new { Sid = student.Sid })
    @Html.ActionLink("Edit", "EditStudent", new { Sid = student.Sid })
    @Html.ActionLink("Delete", "DeleteStudent", new { Sid = student.Sid },
                    new { onclick = "return confirm('Are you sure of deleting the record?')"})
</td>
</tr>
}
<tr><td colspan="6" align="center">@Html.ActionLink("Add New Student", "AddStudent")</td></tr>
</table>

```

Generate a view for **DisplayStudent** action method and while adding the View, choose the **Template** as **Empty**, **Model Class** as **Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Student Details</h2>
<table border="1" align="center">
<tr>
<td rowspan=4><img src='~/Uploads/@Model.Photo' width="200" height="200" alt="No Image" /></td>
<td>Sid: @Model.Sid</td>
</tr>
<tr><td>Name: @Model.Name</td></tr>
<tr><td>Class: @Model.Class</td></tr>
<tr><td>Fees: @Model.Fees</td></tr>
<tr><td colspan="2" align="center">@Html.ActionLink("Back to Student Details", "DisplayStudents")</td></tr>
</table>

```

Generate a view for **AddStudent** action method and while adding the View, choose the **Template** as **Empty**, **Model Class** as **Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
@using (Html.BeginForm("AddStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{

```

```

<div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid)</div>
<div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
<div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
<div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
<div>@Html.LabelFor(S => S.Photo)<br /><input type="file" name="selectedFile" /></div>
<div>
    <input type="submit" value="Save" name="btnSave" />
    <input type="reset" value="Reset" name="btnReset" />
</div>
}
@Html.ActionLink("Back to Student Details", "DisplayStudents")

```

Generate a view for **EditStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class as Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Student Details</h2>
@using (Html.BeginForm("UpdateStudent", "Student", FormMethod.Post, new { enctype="multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Sid)<br />@Html.TextBoxFor(S => S.Sid, new { @readonly = "true" })</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Class)<br />@Html.TextBoxFor(S => S.Class)</div>
    <div>@Html.LabelFor(S => S.Fees)<br />@Html.TextBoxFor(S => S.Fees)</div>
    <div>
        @Html.LabelFor(S => S.Photo)<br />
        <img src='~/Uploads/@Model.Photo' width="100" height="100" alt="No Image" style="border:dashed red" />
        <input type="file" name="selectedFile" />
    </div>
    <div>
        <input type="submit" value="Update" name="btnUpdate" />
        @Html.ActionLink("Cancel", "DisplayStudents")
    </div>
}

```

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 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** supported **4** 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**, **VaryByCustom** 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 Filter** 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.

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.

To test filters, create a new **Table** in our “**MVCDB**” **Database** with the name **Customer** as following, and insert some records into the table:

```
Create Table Customer (Custid Int Primary Key, Name Varchar(50), Balance Money, City Varchar(50), Status Bit Default 1 Not Null)
```

Create a new “**ASP.NET Web Application**” project naming it as “**MVCFilters**”; choose “**Empty Project Template**”, select “**MVC**” CheckBox and click on “**Create**” button. First host the Web Application on IIS and then under Model’s folder add “**ADO.NET Entity Data Model**” naming it as “**TestEF**”, choose “**EF Designer from database**” (**DB First**), provide the connection details for our “**MVCDB**” Database and choose “**Department**”, “**Employee**” and “**Customer**” tables. Add a controller under **Controller’s** folder naming it as “**HomeController**”, delete existing content under the class and write the code:

```
using System.Web.UI;
using MVCFilters.Models;
public class HomeController : Controller
{
    MVCDBEntities dc = new MVCDBEntities();

    #region ChildActionOnly Filter
    public ViewResult DisplayDepts()
    {
        return View(dc.Departments);
    }
    public ViewResult DisplayEmpsByDept(int Did)
    {
        var Emps = from E in dc.Employees where E.Did == Did select E;
        return View(Emps);
    }
    #endregion

    #region OutputCache Filter
    public ViewResult DisplayCustomers1()
    {
        return View(dc.Customers);
    }
    [OutputCache(Duration = 45, Location = OutputCacheLocation.Server)]
    public ViewResult DisplayCustomers2()
    {
        return View(dc.Customers);
    }
}
```

```

[OutputCache(Duration = 45, Location = OutputCacheLocation.Server, VaryByParam = "City")]
public ViewResult DisplayCustomers3(string City)
{
    return View(from C in dc.Customers where C.City == City select C);
}
[OutputCache(Duration = 45, Location = OutputCacheLocation.Server, VaryByCustom = "browser")]
public ViewResult DisplayCustomers4()
{
    return View(dc.Customers);
}
public ViewResult DisplayCustomers5()
{
    return View(dc.Customers);
}
#endregion

#region ValidateInput Filter
public ViewResult GetComments()
{
    return View();
}
[HttpPost]
public string GetComments(string txtComments)
{
    return txtComments;
}
#endregion

#region ValidateAntiForgeryToken Filter
public ViewResult AddEmployee()
{
    return View();
}
[HttpPost]
public string AddEmployee(Employee Emp)
{
    Emp.Status = true;
    dc.Employees.Add(Emp);
    dc.SaveChanges();
    return "Record Inserted";
}
#endregion

#region HandleError Filter
public ViewResult DivideNums()
{

```

```

        return View();
    }
    [HttpPost]
    public string DivideNums(int num1, int num2)
    {
        int result = num1 / num2;
        return "Result is: " + result;
    }
    #endregion
}

```

Add a view with the name “`DisplayEmpsByDept.cshtml`”, selecting layout `Checkbox` and write the below code in it by deleting the existing code in the file:

```

@model IEnumerable<MVCFilters.Models.Employee>
 @{
    Layout = null;
}


| @Html.DisplayNameFor(E => E.Eid) | @Html.DisplayNameFor(E => E.Ename) | @Html.DisplayNameFor(E => E.Job) | @Html.DisplayNameFor(E => E.Salary) | @Html.DisplayNameFor(E => E.Status) |
|----------------------------------|------------------------------------|----------------------------------|-------------------------------------|-------------------------------------|
|                                  |                                    |                                  |                                     |                                     |


```

Add a view with the name “`DisplayDepts.cshtml`”, selecting layout `Checkbox` and write the below code in it by deleting the existing code in the file:

```

@model IEnumerable<MVCFilters.Models.Department>
 @{
    ViewBag.Title = "Display Depts";
}





```

```

<caption>Department Details</caption>
<tr>
    <th>@Html.DisplayNameFor(D => D.Did)</th>
    <th>@Html.DisplayNameFor(D => D.Dname)</th>
    <th>@Html.DisplayNameFor(D => D.Location)</th>
    <th>Employees</th>
</tr>
@foreach (var Dept in Model)
{
    <tr>
        <td>@Html.DisplayFor(D => Dept.Did) </td>
        <td>@Html.DisplayFor(D => Dept.Dname) </td>
        <td>@Html.DisplayFor(D => Dept.Location) </td>
        <td>@Html.Action("DisplayEmpsByDept", new { Did = Dept.Did })</td>
    </tr>
}
</table>

```

ChildActionOnly: in the above case “**DisplayDepts**” view is internally invoking “**DisplayEmpsByDept**” action method so under each “Department” it will display the “Employees” corresponding to that “Department”. If required, we can also directly call “**DisplayEmpsByDept**” action method from browser as below and access “Employee” data corresponding to a particular “Department”.

<http://localhost/MVCFilters/Home/DisplayEmpsByDept/10>

If we want to restrict accessing the “**DisplayEmpsByDept**” action method directly from a browser as above we need to decorate the method with **[ChildActionOnly]** attribute, so that we are not authorized to call that method directly and should be invoked only as a child action. To test that go to “**HomeController**” and add the “**ChildActionOnly**” attribute to “**DisplayEmpsByDept**” action method as following:

```

[ChildActionOnly]
public ViewResult DisplayEmpsByDept(int Id)
{
    var Emps = from E in dc.Employees where E.Did == Id select E;
    return View(Emps);
}

```

Note: now if we try to access “**DisplayEmpsByDept**” action method directly from a browser we get an **error** and to test that try calling the method directly from the browser.

Add a view for “**DisplayCustomers1**” action method, choosing the layout **CheckBox** and write the below code in it by deleting all the existing content in the file:

```

@model IEnumerable<MVCFilters.Models.Customer>
 @{
    ViewBag.Title = "Display Customer";
 }

```

```

<table align="center" border="1">
<tr>
<td colspan="5">
<h3 style="text-align:center">Request Processed Time: @DateTime.Now.ToString("T")</h3></td>
</tr>
<tr>
<th>@Html.DisplayNameFor(C => C.Custid)</th>
<th>@Html.DisplayNameFor(C => C.Name)</th>
<th>@Html.DisplayNameFor(C => C.Balance)</th>
<th>@Html.DisplayNameFor(C => C.City)</th>
<th>@Html.DisplayNameFor(C => C.Status)</th>
</tr>
@foreach (var Customer in Model)
{
<tr>
<td>@Html.DisplayFor(C => Customer.Custid)</td>
<td>@Html.DisplayFor(C => Customer.Name)</td>
<td align="right">@Html.DisplayFor(C => Customer.Balance)</td>
<td>@Html.DisplayFor(C => Customer.City)</td>
@if (Customer.Status == true)
{
<td>Active</td>
}
else
{
<td>In-Active </td>
}
</tr>
}
</table>

```

Note: same as the above create views for the remaining 4 “DisplayCustomers” action methods also with the same code.

OutputCache: in the above case “DisplayCustomers1” is not cached, so whenever we refresh the View the Processed Time on top of the view will change which provides clarity for us that every time the View is refreshed the page gets re-processed. Rest of the 3 “DisplayCustomers” action methods are cached for “45” seconds because we specified that in the “Duration” property, so in that period if a new request is sent to those action methods it will display the cached output of the View only without processing the whole View again and again.

Location attribute is used on “DisplayCustomers2” action method to specify the cache copy should be created on the server because the default value is “Any” and in this case server decides where to store the cache copy, but when changed to Server then cache copy will be stored on the Server Machine only.

VaryByParam attribute is used on “DisplayCustomers3” action method to specify that there should be a separate cache copy created for each parameter value that is sent to the “City” attribute.

<http://localhost/MVCFilters/Home/DisplayCustomers3?City=Bengaluru>
<http://localhost/MVCFilters/Home/DisplayCustomers3?City=Chennai>
<http://localhost/MVCFilters/Home/DisplayCustomers3?City=Delhi>

VaryByCustom attribute is used on “**DisplayCustomers4**” action method to specify that there should be a separate cache copy created for each type of browser i.e., 1 copy for Edge and 1 copy for IE, etc.

Without specifying cache details on the top of an action method we can use “**CacheProfiles**” i.e., by storing the details of cache under “**Web.config**” file and then we can use them on action methods.

To test this, go to “Web.config” file and write the following code under <system.web> tag:

```
<caching>
  <outputCacheSettings>
    <outputCacheProfiles>
      <add name="MyCacheProfile" duration="45" />
    </outputCacheProfiles>
  </outputCacheSettings>
</caching>
```

Now on the top of “DisplayCustomers5” action method specify the “Cache Profile” details as following:

```
[OutputCache(CacheProfile = "MyCacheProfile")]
public ViewResult DisplayCustomers5()
```

Add a View to “GetComments” action method, choosing the layout CheckBox and write the below code in it:

```
@using(Html.BeginForm())
{
  <div>
    @Html.Label("Comments:")<br />@Html.TextArea("txtComments")
  </div>
  <div>
    <input type="submit" value="Submit" name="btnComment" />
  </div>
}
```

ValidateInput: Run the above view, enter some input into the **TextArea** and click on the “**Submit**” button which will display the output on browser. Whereas if we try to enter the comments with any **Html tags**, for example “**Hello World**” we get an error as following: “**A potentially dangerous Request.Form value was detected from the client (txtComments=Hello).**”, because by default “**ValidateInput**” filters value is “**true**” only which will not allow any **Html Elements**, whereas if we want to allow **Html Elements** as input then set the “**ValidateInput**” filter value as “**false**” for “**GetComments**” Post action method as following:

```
[HttpPost, ValidateInput(false)]
public string GetComments(string txtComments)
```

Add a View to “AddEmployee” action method choosing the layout CheckBox and write the below code in it:

```
@model MVCFilters.Models.Employee
```

```

@using (Html.BeginForm())
{
    <div>@Html.LabelFor(E => E.Ename)<br />@Html.TextBoxFor(E => E.Ename)</div>
    <div>@Html.LabelFor(E => E.Job)<br />@Html.TextBoxFor(E => E.Job)</div>
    <div>@Html.LabelFor(E => E.Salary)<br />@Html.TextBoxFor(E => E.Salary)</div>
    <div>@Html.LabelFor(E => E.Did)<br />@Html.TextBoxFor(E => E.Did)</div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" />
    </div>
}

```

ValidateAntiForgeryToken: run the above view to add a new Employee. The problem in the above is if we use the “View Page Source” option of browser, copy the content from <form> to </form> tag, paste it in a new file and change the “action” attribute value of <form> tag from “MVCFilters/Home/AddEmployee” to <http://localhost/MVCFilters/Home/AddEmployee>, save the file as “AddEmployee.html” and when we open the “AddEmployee.html” file in browser using its physical path and try to add a new Employee, still it adds a record into the table. To restrict this, we need to use the “ValidateAntiForgeryToken” filter and to test it do the following:

Step 1: re-write the AddEmployee, HttpPost method as following:

```

[HttpPost, ValidateAntiForgeryToken]
public string AddEmployee(Employee Emp)

```

Step 2: in “AddEmployee.cshtml” file write “@Html.AntiForgeryToken()” statement inside the “<form>” tag, so that if anyone copies the source code and try to use it for adding a new record i.e., same as we discussed above will not be able to invoke the action method to add a record into the table.

Add a view to “DivideNums” action method choosing the layout CheckBox and write the below code in it:

```

@using (Html.BeginForm())
{
    <div>@Html.Label("Enter 1st Number: ")<br />@Html.TextBox("num1")</div>
    <div>@Html.Label("Enter 2nd Number: ")<br />@Html.TextBox("num2")</div>
    <div>
        <input type="submit" value="Divide" name="btnDiv" />
        <input type="reset" value="Reset" />
    </div>
}

```

HandleError: in the above view if we give the value of 2nd number as zero or enter any non-integer values in both Textbox’s or if the entered integer value is beyond the size of an integer, Exception gets raised in the program and display the error details on browser screen. To avoid the problem of displaying error details of the “Exception” on-screen do the following:

Step 1: add a view in Shared folder of views with the name “Error.cshtml” without choosing a layout and write the below code under its “<div>” tag:

```
<h1 style="background-color:lightgoldenrodyellow;color:orangered;text-align:center;text-decoration:underline">
    Server Error Page
</h1>
<h3>An error occurred while processing your request. Re-check your input or contact customer support.</h3>
```

Step 2: go to “Web.config” file and “On” the “custom errors” option by writing the following code under “<system.web>” tag:

```
<customErrors mode="On" />
```

Step 3: now in the HomeController class re-write the “DivideNums” HttpPost method as below:

```
[HttpPost, HandleError]
```

```
public string DivideNums(int num1, int num2)
```

Now if we get any Internal Server Errors (Exceptions) in “DivideNums” methods, it will automatically redirect to “Error.cshtml”.

Now in “HomeController” add a new action method “ShowView” as following:

```
[HandleError]
public ViewResult ShowView()
{
    return View();
}
```

For the above action method, we have not created any view, so if we try to call this action method from a browser it will raise an error and redirect to “Error.cshtml” view only because we have decorated the method with “HandleError” attribute.

Now if we try to call any action method which is not existing in the application then we get an “on screen error” again without displaying “Error.cshtml” with Http Status Code 404 because currently “HandleError” attribute will handle only the errors with Status Code 5XX series only i.e., Internal Server Errors or Exceptions, whereas if we want to handle client errors with their appropriate “Http Status Codes” do the following:

Step 1: add a new controller in Controller’s folder naming it as ErrorController and write the below code in it:

```
public class ErrorController : Controller
{
    public ActionResult BadRequest()
    {
        return View();
    }
    public ActionResult Unauthorized()
    {
        return View();
    }
    public ActionResult PaymentRequired()
    {
        return View();
    }
}
```

```

public ActionResult Forbidden()
{
    return View();
}
public ActionResult NotFound()
{
    return View();
}
}

```

Step 2: now under **Shared** folder of **Views** folder, add a new view naming it as “**NotFound.cshtml**” without choosing a layout and write the below code under the “**<div>**” tag:

```

<h1 style="background-color:lightgoldenrodyellow;color:orangered;text-align:center;text-decoration:underline">
    Page Not Found
</h1>
<h3>The requested page is either not available or moved to a different location or it's under construction or
    maintanance.</h3>

```

Step 3: now go to “Web.config” file and re-write the custom errors element as following:

```

<customErrors mode="On">
    <error statusCode="400" redirect="/Error/BadRequest"/>
    <error statusCode="401" redirect="/Error/Unauthorized"/>
    <error statusCode="402" redirect="/Error/PaymentRequired"/>
    <error statusCode="403" redirect="/Error/Forbidden"/>
    <error statusCode="404" redirect="/Error/NotFound"/>
</customErrors>

```

Note: same as the above we add a separate error page for each **Http Status Code** as per your requirements.

Handling Errors Controller Level: in the above case to handle errors we are applying “**HandleError**” attribute for each **Action** method and without doing that we can apply “**HandleError**” attribute directly on the **Controller** class so that it applies to all **Action** methods under that **Controller** and to test that delete “**HandleError**” attribute we have used on “**Http Post**” action method of “**DivideNums**” and “**ShowView**” action method, and then use it on the **Controller** class i.e., “**HomeController**” as following:

```

[HandleError]
public class HomeController : Controller

```

Handling Errors Globally: in the above approach we are applying “**HandleError**” attribute on **Controller** class which must be done on every **Controller** in the project to handle the **errors**. To avoid this, we can apply filters **globally** to the whole application or project and to do that do the following:

Step 1: Add a Code File under “App_Start” folder, naming it as “FilterConfig.cs” and write the below code in it:

```

using System.Web.Mvc;
namespace MVCFilters

```

```

{
    public class FilterConfig
    {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters)
        {
            filters.Add(new HandleErrorAttribute());
        }
    }
}

```

Step 2: go to “Global.asax.cs” file in the project and write the below statement in “Application_Start” method:

```
FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
```

Note: with the above action, errors that occur in all the **action methods** of all the **controllers** get handled.

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 and 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

Anonymous Authentication: This gives users access to the public areas of our **Application** without asking them for a **username** or **password**. By default, the “**IUSR**” account, which was introduced in **IIS 7.0**, is used to allow anonymous access. By default, IIS uses **Anonymous authentication** for every **Application**. To test this, open “**IIS Manager**” and in the **LHS** we find “**Connections Panel**” and under that we find the “**Server Name**” (which will be same as our computer name), expand it and under that we find a node “**Sites**”, expand it and under that we find a node “**Default Web Site**”, expand it and under that we find our project “**MVCFilters**” (because we hosted it in **IIS** from our **Visual Studio** by creating a **Virtual Directory**), select it and then in the **RHS** we find “**Authentication**” icon, click on it which display the various authentication modes available and under that we can see “**Anonymous Authentication**” is **enabled**. Right click on the “**Anonymous Authentication**” and select “**Edit**” which display the user account i.e., “**IUSR**” which is used for “**Anonymous Authentication**” as below:



Windows Authentication: we use **Windows authentication** when our “**IIS Server**” runs on a corporate network that is using **Microsoft Active Directory** service domain identities or other **Windows** accounts to identify users. **Windows**

Authentication is a secure form of **Authentication** because the **username** and **password** are hashed before being sent across the network. When you enable **Windows Authentication**, the client browser sends a strongly hashed version of the password in a **cryptographic exchange** to the **Web Server**. This **Authentication** mode is best suitable for **Intranet Applications** only.

Implementing Windows Authentication using Authorize Attribute: Create a new “ASP.NET Web Application” project, naming it as “MVCAppWithWindowsAuth”, choose **Empty Project Template**, select “**MVC CheckBox**” and click on the “Create Button”. Now host the project under IIS by creating a **Virtual Directory** and then open **IIS Manager** select our “MVCAppWithWindowsAuth” project under the “Default Web Site” and now on the RHS select the option “**Authentication**” will displays **Authentication** options and in that if you notice “**Anonymous Authentication**” is enabled by default, so right click on it and select the option “**Disable**” and then right click on the “**Windows Authentication**” and select the option “**Enable**” because by default it is “**Disabled**”.

Add a controller under **Controller’s** folder naming it as **HomeController**, add a view to **Index** action method choosing a **layout**, delete the existing content in it and write the below code over there:

```
@{ ViewBag.Title = "Site - Home Page"; }
<h2>Site Home Page</h2>
<h3>
    @Html.ActionLink("Manager Home Page", "Home", "Manager") <br />
    @Html.ActionLink("Manager Help Page", "Help", "Manager") <br />
    @Html.ActionLink("Staff Home Page", "Home", "Staff") <br />
    @Html.ActionLink("Staff Help Page", "Help", "Staff")
</h3>
```

Add another 2 controllers under the Controllers folder with the names “**ManagerController**” and “**StaffController**”, delete the existing **Index Action** method under the classes and write the below code in the 2 new controller classes:

```
public class ManagerController : Controller
{
    public ActionResult Home()
    {
        return View();
    }
    public ActionResult Help()
    {
        return View();
    }
}

public class StaffController : Controller
{
    public ActionResult Home()
    {
        return View();
    }
}
```

```

public ActionResult Help()
{
    return View();
}
}

```

Now add views to all the above action methods, choosing a layout and write the following code under them by deleting the existing code in it:

Code under Manager's Home - Action Method's view:

```

@{ ViewBag.Title = "Manager - Home Page"; }
<h2>Manager Home Page</h2>
<h3>This page is accessible only to Managers.</h3>
<h4>Current User: @HttpContext.Current.User.Identity.Name</h4>
<br />
Back to site @Html.ActionLink("home page", "Index", "Home")

```

Code under Manager's Help - Action Method's view:

```

@{ ViewBag.Title = "Manager - Help Page"; }
<h2>Manager Help Page</h2>
<h3>This page is accessible to everyone.</h3>
<h4>Current User: @HttpContext.Current.User.Identity.Name</h4><br />
Back to site @Html.ActionLink("home page", "Index", "Home")

```

Code under Staff Home - Action Method's view:

```

@{ ViewBag.Title = "Staff - Home Page"; }
<h2>Staff Home Page</h2>
<h3>This page is accessible only to Staff Members.</h3>
<h4>Current User: @HttpContext.Current.User.Identity.Name</h4><br />
Back to site @Html.ActionLink("home page", "Index", "Home")

```

Code under Staff Help - Action Method's view:

```

@{ ViewBag.Title = "Staff - Help Page"; }
<h2>Staff Help Page</h2>
<h3>This page is accessible to everyone.</h3>
<h4>Current User: @HttpContext.Current.User.Identity.Name</h4><br />
Back to site @Html.ActionLink("home page", "Index", "Home")

```

Now run **Home Controller's - Index Action** method which will display links for navigating to **Manager Home Page**, **Manager Help Page**, **Staff Home Page** and **Staff Help Page**, and once we click on any link those pages get opened without any restrictions because currently the site is running in **Anonymous Authentication** whereas if we want **Manager Home Page** to be accessible only to **Manager's** and **Staff Home Page** to be accessible only to **Staff**, we can do that by using "**Authorize**" filter.

To do that first we need to create new user accounts on our machine and to create new user accounts do the following, open "**Computer Management**" window and in that we find "**Local Users and Groups**" in **LHS**, expand

it and select Users which display users on your computer. To create a new user account right click on “Users” and select “New User” option which opens a window and in that enter “UserName”, “Password” and “Confirm Password” values, uncheck the CheckBox “User must change password at next logon” and check the CheckBox’s “User cannot change password” and “Password never expires”, and then click “Create” button to create the user. Follow this process and create 8 new user accounts with the name Manager1, Manager2, Staff1, Staff2, Staff3, Staff4, Staff5 and Staff6.

Now go to ManagerController class and use Authorize attribute on Home Action method as following:

```
[Authorize(Users = "Server\\Manager1, Server\\Manager2")]
public ActionResult Home()
```

Same as above go to StaffController class and use Authorize attribute on Home Action method as following:

```
[Authorize(Users = "Server\\Staff1, Server\\Staff2, Server\\Staff3, Server\\Staff4, Server\\Staff5, Server\\Staff6")]
public ActionResult Home()
```

Note: in the above case **Server** is my machine name and same as that you need to write your machine name there.

Now use the switch user option in your machine, log in by using any new user account and based on the user you have logged in those **Index Action Methods** will be accessible whereas rest of other **Action Methods** in the controller will run in **Anonymous Authentication**.

Applying Authorize Filter - Controller Level: if required we can use **Authorize Filter - Controller level** and in that case all the **Action Methods** of that controller will be accessible to valid users only and we do that as following:

```
[Authorize(Users = "Server\\Manager1, Server\\Manager2")]
public class ManagerController : Controller
{
}

[Authorize(Users = "Server\\Staff1, Server\\Staff2, Server\\Staff3, Server\\Staff4, Server\\Staff5, Server\\Staff6")]
public class StaffController : Controller
{
}
```

Allowing Anonymous Authentication to some Action Methods in case we apply Authorize Filter Controller level:

when we apply **Authorize Filter** in controller level all action methods in that controller will be running under **Authentication Mode** and if we want any action method to be running in **Anonymous Authentication** mode we need to use “**AllowAnonymous**” filter on those **Action Methods**. In our above example we want **Help Action Methods** of **ManagerController** and **StaffController** to run in **Anonymous Authentication** and to do that add “**AllowAnonymous**” attribute to those methods in both controllers as following:

```
[AllowAnonymous]
public ActionResult Help()
```

Role Based Authentication: in the above case we have given authentication to users by specifying their names in a comma-separated list but if at all we have multiple users like 5 **Managers** and 20 **Staff Members** we need to specify the complete list of members under **Authorize Filter** and to avoid this we have an option of **Role Based Authentication**.

User Groups: under the Operating System apart from Users we also find Groups and to check this open “Computer Management” window, expand “Local Users and Groups” option which displays Users and Groups below, when we select users, it displays existing users and when we select groups’ displays existing groups. Every group is a collection of users, and every user must be members of some group, to check this, select users, double click on any username which displays a window “Administrator Properties” and in that we find a tab “Member Of” click on it which displays the list of groups to which this user belongs to and these groups are what we call as roles in our MVC Applications.

We can also create our own groups or roles under O.S. and to do that right click on Groups, select “New Group” which opens a Window, enter group name in it as “ManagerGroup”, click on “Add” button which opens a window in that under the last TextBox enter “Manager1” click “Check Names” button which displays “ComputerName\Manager1” and repeat the same process to add “Manager2” also to this group. Now follow the same process to create “StaffGroup” and add the 6 Staff members to this group.

Now replace “Users” property under Authorize filter to “Roles” and specify the role name which should be as below for ManagerController and StaffController:

```
[Authorize(Roles = "ManagerGroup")]
public class ManagerController : Controller
```

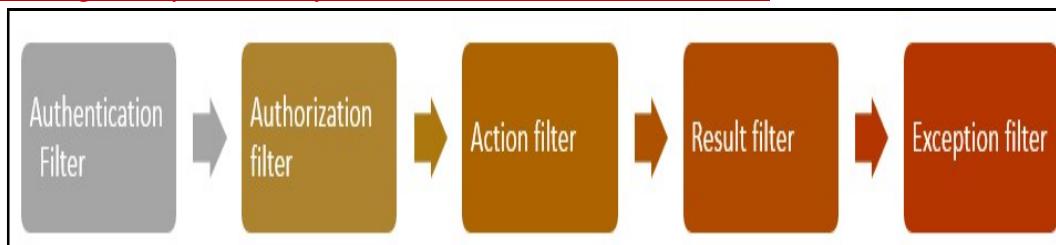
```
[Authorize(Roles = "StaffGroup")]
public class StaffController : Controller
```

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 below:

- 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:



To create a custom filter, we need to perform the following tasks:

- Define a class inheriting from the pre-defined class “FilterAttribute”.
- Override any of the methods that are present in “FilterAttribute” class as per our requirements (optional).
- Implement any of the 5 filter “Interfaces” based on the type of Filter we are implementing, for example if we are implementing Authentication Filters, we should implement IAuthenticationFilter interface and if we are implementing Authorization Filters, we should implement IAuthorizationFilter interface and so on.
- Apply the new filter on the Controller class or Action method where we want to use it.

Implementing Forms Authentication by defining a Custom Filter:

Create a new “ASP.NET Web Application” project naming it as “MVCAppWithFormsAuth”, choose Empty Project Template, select “MVC Check Box” and click on the “Create Button”. Host the project on IIS by creating a Virtual Directory. Add a controller in Controller’s folder naming it as “HomeController”, add a view to Index action method by choosing the layout and write below code in it:

```
<h1 style="background-color:red;color:yellow;text-align:center">Home Page</h1>
<h4>This is Home Page of the site.</h4>
```

Right now, we can run the Index View or any other Views in the site directly because Anonymous Authentication is enabled, but if we want to implement Forms Authentication, so that only Registered Users can access the pages in site, we need to do that by following the below process:

Step 1: create a new table under our “MVCDB” database as following:

```
Create Table Users (UserId Varchar(20) Primary Key, Name Varchar(50), Password Varchar(16), Email Varchar(50),
Mobile Varchar(10), Status Bit Not Null Default 1)
```

Step 2: add Ado.Net Entity Data Model (DB First) in the “Models” folder naming it as “TestEF”, choose “EF Designer from database” (DB First), provide the connection details for our “MVCDB” Database and choose “Users” table which will create the necessary classes for working with data and here the Model class representing “Users” table will be having the name as “User”.

Step 3: now we need to design Register and Login forms which requires data validations and to validate the data we need to use data annotations. To do that let’s define 2 new classes “RegisterModel” and “LoginModel”, and apply data annotations on those classes, so that we can use those classes for designing the Views by performing Model Binding and we call this as MVVM (Model View – View Model) architecture.

Add 2 new classes in Model’s folder naming them as “RegisterModel.cs” and “LoginModel.cs” and write the below code under the 2 classes:

```
using System.ComponentModel.DataAnnotations;
public class RegisterModel
{
    [Display(Name = "User Id")]
    [Required(ErrorMessage = "Can't leave the field empty.")]
    [RegularExpression("[A-Za-z0-9]{6,20}", ErrorMessage = "User Id value is invalid.")]
    public string UserId { get; set; }
```

```

[Required(ErrorMessage = "Can't leave the field empty.")]
[RegularExpression("[A-Za-z\\s]{3,50}", ErrorMessage = "Name value is invalid.")]
public string Name { get; set; }

[DataType(DataType.Password)]
[Required(ErrorMessage = "Can't leave the field empty.")]
[RegularExpression("[A-Z]{1}[a-z0-9@#$%_-]{7,15}", ErrorMessage = "Password is invalid.")]
public string Password { get; set; }

[Display(Name = "Confirm Password")]
[DataType(DataType.Password)]
[Compare("Password")]
public string ConfirmPassword { get; set; }

[Required(ErrorMessage = "Can't leave the field empty.")]
[DataType(DataType.EmailAddress)]
public string Email { get; set; }

[Required(ErrorMessage = "Can't leave the field empty.")]
[RegularExpression("[6-9]\\d{9}", ErrorMessage = "Phone is invalid.")]
public string Mobile { get; set; }

}

using System.ComponentModel.DataAnnotations;
public class LoginModel
{
    [Display(Name = "User Id")]
    [Required(ErrorMessage = "Can't leave the field empty.")]
    [RegularExpression("[A-Za-z0-9]{6,20}", ErrorMessage = "User Id value is invalid.")]
    public string UserId { get; set; }

    [DataType(DataType.Password)]
    [Required(ErrorMessage = "Can't leave the field empty.")]
    [RegularExpression("[A-Z]{1}[a-z0-9@#$%_-]{7,16}", ErrorMessage = "Password is invalid.")]
    public string Password { get; set; }
}

```

Step 4: add a new Controller under Controller's folder naming it as “**AccountController**” and write the below code in it by deleting all the existing code under the class:

```

using MVCAccountWithFormsAuth.Models;
public class AccountController : Controller
{
    MVCDBEntities dc = new MVCDBEntities();
    public ViewResult Register()
    {

```

```

        return View();
    }

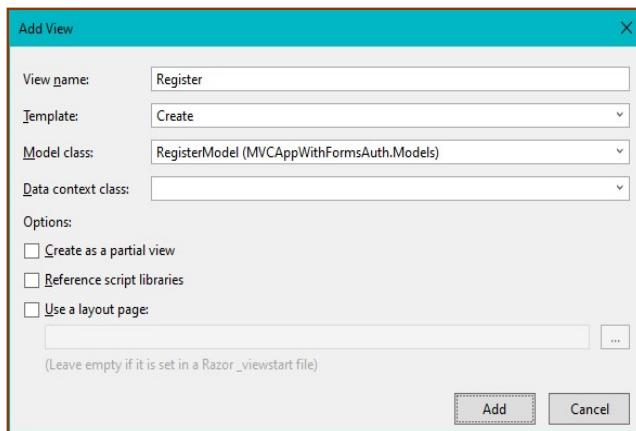
[HttpPost, ValidateAntiForgeryToken]
public ActionResult Register(RegisterModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    else
    {
        User user = new User { UserId = model.UserId, Name = model.Name, Password = model.Password,
            Email = model.Email, Mobile=model.Mobile, Status = true };
        dc.Users.Add(user);
        dc.SaveChanges();
        return RedirectToAction("Login");
    }
}

public ViewResult Login()
{
    return View();
}

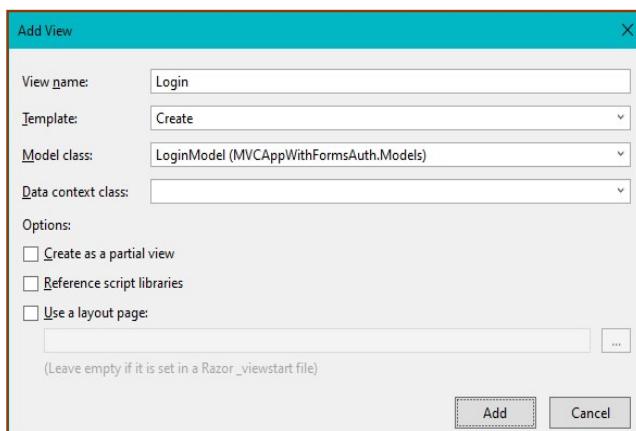
[HttpPost, ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    else
    {
        var user = from u in dc.Users where u.UserId == model.UserId && u.Password == model.Password &&
            u.Status == true select u;
        if(user.Count() == 0)
        {
            ModelState.AddModelError("", "Invalid Credentials");
            return View(model);
        }
        else
        {
            Session["UserKey"] = Guid.NewGuid();
            return RedirectToAction("Index", "Home");
        }
    }
}
}

```

Step 5: Add a view to Registration Action method by choosing “Create” under “Template:” DropDownList, “RegisterModel (MVCAppWithFormsAuth.Models)” under “Model class:” DropDownList, uncheck all the CheckBox’s on the screen and click on “Add” button:



Same as the above add a View to Login Action method by choosing “Create” under “Template:” DropDownList, “LoginModel (MVCAppWithFormsAuth.Models)” under “Model class:” DropDownList, uncheck all the CheckBox’s on the screen and click on “Add” Button:



Install “Bootstrap”, “jQuery”, “jQuery.Validation”, and “Microsoft.jQuery.Unobtrusive.Validation” libraries in our project using “Nuget Package Manager” and then drag and drop the following files into the “`<head>`” section of “Register.cshtml” and “Login.cshtml”: “jquery-3.7.0.min.js”, “jquery.validate.min.js”, “jquery.validate.unobtrusive.min.js” from Script’s folder and “bootstrap.min.css” from Content’s folder.

Now we can create new user accounts by using “Register View” and login into the application by using “Login View” and once we successfully login, it will take to “Home Controller’s - Index Page”, but we can open this Page even by directly calling it also because, right now we did not apply “Authentication” in our application.

Step 6: to apply **Authentication** let us define a custom “Authentication Filter” class under the Project. To do that, first add a new folder under the project naming it as “Filters” and under that folder add a class naming it as “AuthenticateFilter.cs” and write the following under the class:

```

using System.Web.Mvc;
using System.Web.Mvc.Filters;
public class AuthenticateFilter : FilterAttribute, IAuthenticationFilter
{
    public void OnAuthentication(AuthenticationContext filterContext)
    {
        if (string.IsNullOrEmpty(Convert.ToString(filterContext.HttpContext.Session["UserKey"])))
        {
            filterContext.Result = new UnauthorizedResult();
        }
    }
    public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)
    {
        if (filterContext.Result == null || filterContext.Result is UnauthorizedResult)
        {
            filterContext.Result = new RedirectResult("~/Account/Login");
        }
    }
}

```

Note: as explained above every filter should inherit from “**FilterAttribute**” class and implement any of the 5 pre-defined filter **Interfaces** based on the type of filter we want to implement. In the current context we are implementing an **Authentication Filter**, so we need to implement “**IAuthenticationFilter**” interface and write the logic for the 2 abstract methods of the **interface** and those are: “**OnAuthentication**” and “**OnAuthenticationChallenge**”.

Step 7: now go to “**HomeController.cs**” file and apply the “**AuthenticateFilter**” attribute on the class so that Authentication comes into picture when we try to access the Action methods of “**HomeController**” class and to that first import the namespace “**MVCAppWithFormsAuth.Filters**” and then apply the filter on “**HomeController**” class as following:

```

[AuthenticateFilter]
public class HomeController : Controller

```

From now when we try to access any Action Method of “**Home Controller**”, first “**AuthenticateFilter**” class will check whether the “**User**” has a “**UserKey**” value which we have given on successful login to each **User** by storing it in a **Session**. If that “**UserKey**” value is **missing** then it will redirect the **User** to “**Login View**” of “**Account Controller**”, so that unless and until a **User** is logged in to the application, **he/she** can’t access any page of “**Home Controller**”.

Note: If we want to apply this behavior to all the other **Controllers** in our Application, use the “**AuthenticateFilter**” attribute on all the **Controller Classes** in the project **except** on “**Account Controller**” class.

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.

To work with Ajax and JQuery, create a new "ASP.NET Web Application" project, naming it as "MVCAppUsingAjax", choose "Empty Project Template", select MVC - Check Box and click on the Create - Button. Add a folder under the project naming it as "Matches", add an "XML File" under this folder naming it as "Score.xml" and write the following code in it:

```
<Match1>
<Score>Match starts at 7:00 PM IST</Score>
</Match1>
```

To work with JQuery, we need to install the JQuery plug-in in our project and to do that open "Nuget Package Manager", search for "JQuery" and install "jQuery by jQuery Foundation, Inc," (current version is 3.7.1), which will add a new folder under the project with the name Scripts and installs all the JQuery files into the folder.

Add a Controller in Controller's folder naming it as "HomeController" and write the below code in the class:

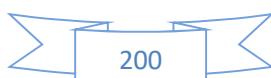
```
using System.Xml.Linq; //For using LINQ to XML
public string GetScore()
{
    //Gets the physical path of the file for the given virtual path
    string physicalPath = Server.MapPath("~/Matches/Score.xml");
    //Loads the XML file into the application from the specified location
    var doc = XElement.Load(physicalPath);
    //Reads the value or content of "Score" element from the xml file which is loaded
    string score = doc.Element("Score").Value;
    return score;
}
```

Now add a view to “**Index**” action method without choosing any layout and drag & drop the **JQuery** file “**jquery-3.7.1.min.js**” from “**Scripts**” folder into the head section and write the below code under “**<body>**” tag by deleting the existing “**<div>**” tag in it:

```
<body onload="LoadData()">
<div style="background-color:lightgreen;color:blueviolet">
    Date & Time: @DateTime.Now.ToString()
</div>
<div style="background-color:yellow;color:red">
    <marquee id="m1" behavior="alternate"></marquee>
</div>
<div style="background-color:coral;color:chartreuse">
    Date & Time: @DateTime.Now.ToString()
</div>
<script>
    function LoadData()
    {
        $.ajax({
            url: "/Home/GetScore",
            type: "get",
            cache: false,
            success: F1,
            error: F2
        });
        window.setTimeout("LoadData()", 1000);
    }
    function F1(responseString)
    {
        $("#m1").html(responseString);
    }
    function F2()
    {
        window.alert("Error in server.");
    }
</script>
</body>
```

Now run the **Index View** and watch the output which will display the current score value that is present inside of “**<Score>**” element under the **XML** file and whenever we change the value under “**<Score>**” element it will automatically display the modified score value with-in the “**<marquee>**” element of the page, because we have called the “**setTimeout**” function with an interval of 1 second (**1000 MilliSeconds**) and all this happens behind the screen without submitting the page to server, by **Ajax**.

Implementing Auto-Complete TextBox using Ajax and JQuery: Add “**ADO.NET Entity Data Model**” item under **Model’s** folder of our previous project i.e., “**MVCAppUsingAjax**”, naming it as “**TestEF**”, choose “**EF Designer from database**”, configure it with “**Northwind**” Database and choose the table “**Products**”. Add a controller under **Controller’s** folder naming it as “**ProductController**” and write the below code in it:



```

using MVCAppUsingAjax.Models;
public class ProductController : Controller
{
    NorthwindEntities dc = new NorthwindEntities();
    public ViewResult DisplayProducts()
    {
        return View(dc.Products);
    }
    [HttpPost]
    public ViewResult SearchProduct(string SearchTerm)
    {
        List<Product> Products;
        if (SearchTerm.Trim().Length == 0)
        {
            Products = dc.Products.ToList();
        }
        else
        {
            //You can implement the Query in any of the below 2 ways:
            Products = (from P in dc.Products where P.ProductName.Contains(SearchTerm) select P).ToList();
            Or
            Products = dc.Products.Where(P => P.ProductName.Contains(SearchTerm)).ToList();
        }
        return View("DisplayProducts", Products);
    }
    public JsonResult GetProducts(string term)
    {
        List<string> Products = dc.Products.Where(
            P => P.ProductName.StartsWith(term)).Select(P => P.ProductName).ToList();
        return Json(Products, JsonRequestBehavior.AllowGet);
    }
}

```

Install “**JQuery.UI.Combined**” (for “**AutoComplete**” functionality) and “**JQuery.UI.Theme.uidarkness**” (for setting a “**Theme**” for “**AutoComplete**” output) plug-ins in our project using “**Nuget Package Manager**”, add a view to “**DisplayProducts**” Action method without choosing any layout and do the following in the View file:

Add the below statement on top of the View:

```
@model IEnumerable<MVCAppUsingAjax.Models.Product>
```

Drag and drop the below “CSS” and “JS” files under the <head> tag:

```
<script src="~/Scripts/jquery-3.6.0.min.js"></script>
<script src="~/Scripts/jquery-ui-1.13.2.min.js"></script>
<link href="~/Content/themes/ui-darkness/jquery.ui.theme.css" rel="stylesheet" />
```

Write the below logic for “AutoComplete” within the <head> tag:

```
<script>
$(function () {
    $("#SearchTerm").autocomplete({
        source: '@Url.Action("GetProducts")'
    });
});
</script>
```

Now write the following code under <div> tag of body:

```
@using (Html.BeginForm("SearchProduct", "Product"))
{
    <center>
        <b>Product Name:</b>
        @Html.TextBox("SearchTerm")
        <input type="submit" id="btnSubmit" value="Search" />
    </center>
}
<table border="1" align="center">
    <caption>Product Details</caption>
    <tr>
        <th>@Html.DisplayNameFor(P => P.ProductID)</th>
        <th>@Html.DisplayNameFor(P => P.ProductName)</th>
        <th>@Html.DisplayNameFor(P => P.CategoryID)</th>
        <th>@Html.DisplayNameFor(P => P.UnitPrice)</th>
        <th>@Html.DisplayNameFor(P => P.UnitsInStock)</th>
    </tr>
    @foreach (var Product in Model)
    {
        <tr>
            <td>@Html.DisplayFor(P => Product.ProductID)</td>
            <td>@Html.DisplayFor(P => Product.ProductName)</td>
            <td>@Html.DisplayFor(P => Product.CategoryID)</td>
            <td>@Html.DisplayFor(P => Product.UnitPrice)</td>
            <td>@Html.DisplayFor(P => Product.UnitsInStock)</td>
        </tr>
    }
</table>
```

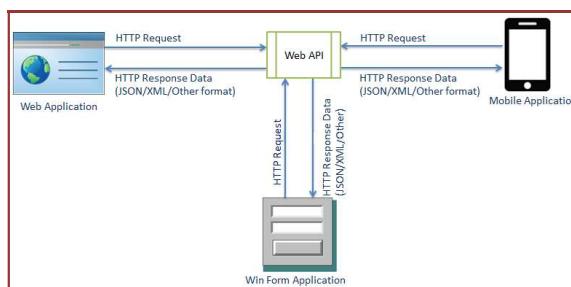
ASP.NET Web API

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 an 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**, **Python** etc. For example, **Twitter’s REST API’s** 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**, and **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 map to the **R (Read)** 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

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

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 all 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.

Creating a Web API Service: Create a new “**ASP.Net Web Application**” project naming it as “**Web ApiService**”, choose “**Empty Project Template**”, select “**Web Api**” CheckBox (which adds all the **folders** and **references** for working with **Web Api**) and click on the “**Create**” Button.

Note: the folder structure here also will be same as the folder structure of an **MVC Project** like **Controllers, Models** etc., but here we will not have **View’s** folder.

Adding a Web Api Controller: to add a Web Api Controller, right click on the Controllers folder and select the option **Add => Controller** and in the window opened, select “**Web Api**” in the **LHS** and then select “**Web Api 2 Controller - Empty**” in **RHS** and name it as “**TestController**” which adds a Controller under the project, but this controller class inherits from “**ApiController**” which is present under the namespace “**System.Web.Http**”. Write the below code in “**TestController**” class:

```
public class TestController : ApiController
{
    //Declaring a List of strings and storing a set of colors in it
    static List<String> Colors = new List<string>()
    {
        "Red", "Blue", "Green", "Purple", "Magenta"
    };

    //Responds for Get (Select) request
    public IEnumerable<String> Get()
    {
        return Colors;
    }
    //Responds for Get (Select with condition) request with a parameter - id
    public string Get(int id)
    {
        return Colors[id];
    }
}
```

```

//Responds for Post (Insert) request
public void Post([FromBody] string color)
{
    Colors.Add(color);
}

//Responds for Put (Update) request
public void Put(int id, [FromBody] string color)
{
    Colors[id] = color;
}

//Responds for Delete (Delete) request
public void Delete(int id)
{
    Colors.RemoveAt(id);
}

```

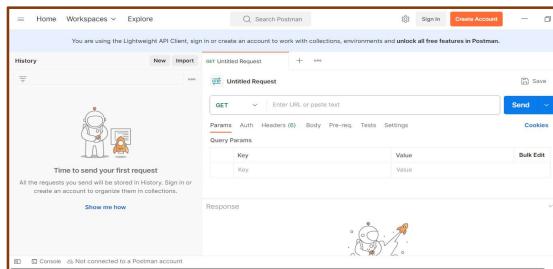
Run the project which will display the Uri <http://localhost:port> in the browser address bar, add “/Api/Test” to it in the last, which will display all the **Colors** on the browser in **XML Format**, because by default every browser will be sending a “Get” request to the Server, so **Get** method got executed displaying all the colors. If we add “/3” after “Test” then also “Get” method only gets executed but the “Get” method which takes “Id” as a parameter will execute and displays a single color in **XML** format.

<http://localhost:port/Api/Test> //For getting all the colors

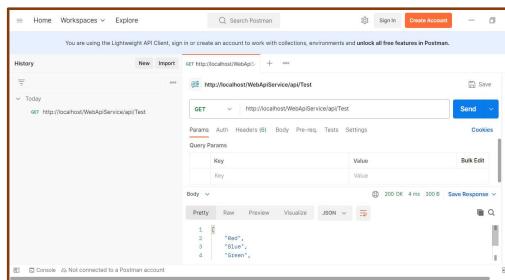
<http://localhost:port/Api/Test/3> //For getting single color with index 3

If we want to test **Post**, **Put** and **Delete** methods we need to use any **Web Debugging Tool** like **Fiddler**, **Postman**, **Swagger** etc., so let’s use **Postman** and to do that first install **Postman** on your machines.

Working with Postman: To test all the operations of our Web API Service using Fiddler, first host the application on **IIS Web Server**, then launch **Postman App.**, click on the “+” button in RHS which opens a **composer window** as below:



Now under the TextBox “Enter URL or paste text”, enter this URL: <http://localhost/WebApiService/Api/Test> and click on **Send** button which will display the output in the below as following:



Testing Post Method: post means insert or add, and to test this go to **composer window**, and there before the address bar we find a **drop-down** and in that **GET** to **POST** and below the address bar select “**body**” tab and in the below we get a **drop-down** with value “**none**” change to “**raw**” which shows another **drop-down** with a value “**Text**” change it to “**JSON**” and enter the value you wanted to add in the **TextArea** below, for example lets enter the value as “**Black**” (**Data should be in double quotes**) and click on **Send** button which will insert or add the value into the resource i.e., List in our case and to confirm that change **POST** to **GET** in drop-down and click on **Send** button again which displays the new value in List.

Note: If we want to test **Put (Update)** and **Delete** we need to make setting in **Web.config** file because when the application is hosted on IIS, it will not support **Put** and **Delete** actions and to resolve this problem do the following:

Step 1: go to **Web.config** file and there we find a node **<system.webServer>** and under that we find **<handlers>** node, write the following statement inside of that **<handlers>** node:

```
<remove name="WebDAV" />
```

Step 2: now under **<system.webServer>** if you find **<modules>** node, add the following under it:

```
<remove name="WebDAVModule"/>
```

Or else add the following:

```
<modules> <remove name="WebDAVModule"/> </modules>
```

Testing Put Method: to test Put or Update, go to **composer window** and in the drop-down change **GET** to **PUT**. To update a value we need to the key to refer to the item we want to update and in our case **Index** is the key, so let's update the 3rd item in the List which means index 2 and to do that add “**/2**” to the URL in the address bar which should now look as following: <http://localhost/Web ApiService/Api/Test/2> and below the address bar select “**body**” tab and we get a **drop-down** below with value “**none**” change to “**raw**” which shows another **drop-down** with a value “**Text**” change it to “**JSON**” and enter the value you wanted to add in the **TextArea** below, for example lets enter the value as “**White**” (**Data should be in double quotes**) and click on **Send** button which will update the value in the resource i.e., List in our case and to confirm that change **Put** to **GET** in drop-down and click on **Send** button again which displays the new value in List.

Testing Delete Method: to test Delete go to **composer windows**, and in the drop-down change **GET** to **Delete**. To delete a value also we need to the key to refer to the item we want to delete and in our case **Index** is the key, so let's delete the 4th item in the List which means index 3 and to do that add “**/3**” to the URL in the address bar which should now look as following: <http://localhost/Web ApiService/Api/Test/3> and click on the **Send** button which will delete the record and to check the output change **Delete** to **GET** in drop-down and also delete “**/3**” in the URL and click on the **Send** button again which displays the new value in List.

Developing a Web API Service to perform CRUD Operations with Database: under the **Models** folder add “**Ado.Net Entity Data Model**”, naming it as “**TestEF**” and choose “**EF Designer from database**” => Configure it with our “**MVCDB**” database and choose the “**Customer**” Table. Add a new “**Web API 2 Controller - Empty**” under the project naming it as “**CustomerController**” and write the below code in the class:

```
using System.Data.Entity;
using Web ApiService.Models;
public class CustomerController : ApiController
{
```

```

MVCDBEntities dc = new MVCDBEntities();
public List<Customer> Get()
{
    return dc.Customers.ToList();
}
public Customer Get(int id)
{
    return dc.Customers.Find(id);
}
public HttpResponseMessage Post(Customer c)
{
    try
    {
        c.Status = true;
        dc.Customers.Add(c);
        dc.SaveChanges();
        return new HttpResponseMessage(HttpStatusCode.Created);
    }
    catch (Exception)
    {
        throw new HttpResponseException(HttpStatusCode.InternalServerError);
    }
}
public HttpResponseMessage Put(Customer c)
{
    try
    {
        Customer obj = dc.Customers.Find(c.Custid);
        if (obj == null)
        {
            return new HttpResponseMessage(HttpStatusCode.NotFound);
        }
        obj.Name = c.Name;
        obj.Balance = c.Balance;
        obj.City = c.City;
        dc.Entry(obj).State = EntityState.Modified;
        dc.SaveChanges();
        return new HttpResponseMessage(HttpStatusCode.OK);
    }
    catch (Exception)
    {
        throw new HttpResponseException(HttpStatusCode.InternalServerError);
    }
}
public HttpResponseMessage Delete(int id)
{

```

```

try
{
    Customer obj = dc.Customers.Find(id);
    if (obj == null) {
        return new HttpResponseMessage(HttpStatusCode.NotFound);
    }
    obj.Status = false;
    dc.Entry(obj).State = EntityState.Modified;
    dc.SaveChanges();
    return new HttpResponseMessage(HttpStatusCode.OK);
}
catch (Exception)
{
    throw new HttpResponseException(HttpStatusCode.InternalServerError);
}
}
}

```

Consuming the WebApi Service using JQuery-Ajax from the same project in which WebApi was developed: add an “MVC Controller” in the current project naming it as “HomeController” and this action will raise errors in the application because we have opened an “Empty Project” and then selected “Web Api” Checkbox but now we are adding an “MVC Controller”. To fix the errors we need to install 2 “Nuget Packages” => “Microsoft.AspNet.Mvc” and “Microsoft.AspNet.Web.Optimization”.

After installing the above 2 packages, open “Global.asax.cs” file and call “RegisterRoutes” method of “RouteConfig” class, and to do that write the below statement under “Application_Start” method in the last line:

```
RouteConfig.RegisterRoutes(RouteTable.Routes);
```

Add a view to the existing “Index” Action method of “HomeController” without choosing any layout. Install “JQuery” into the project using “Nuget Package Manager”, and write the below code in “<head>” section:

```

<script src="~/Scripts/jquery-3.7.1.min.js"></script>
<script>
function GetCustomers() {
    $.ajax({
        url: 'http://localhost/Web ApiService/api/Customer',
        type: 'get',
        datatype: 'json',
        success: BuildTable,
        error: DisplayError
    });
}

function BuildTable(Customers) {
    $('#tblCustomers tbody').empty();
    $.each(Customers, function (Index, Customer) {
        var Status = (Customer.Status) ? 'Active' : 'In-Active';

```

```

        $('#tblCustomers').append('<tr><td align=center>' + Customer.Custid + '</td><td>' + Customer.Name +
        '</td><td align=right>' + Customer.Balance + '</td><td>' + Customer.City + '</td><td align=center>' + Status +
        '</td></tr>');
    });
}

function DisplayError() {
    window.alert("Error on the server, could not load the data.");
}

function GetCustomer() {
    $.ajax({
        url: 'http://localhost/Web ApiService/api/Customer',
        type: 'get',
        datatype: 'json',
        data: {
            'id': $('#Custid').val()
        },
        success: function (Customer) {
            if (Customer == null) {
                window.alert("No customer exists with the given id.");
                $('#frmCustomer').trigger("reset");
                $('#Custid').focus();
            }
            else {
                $('#Name').val(Customer.Name);
                $('#Balance').val(Customer.Balance);
                $('#City').val(Customer.City);
            }
        },
        error: DisplayError
    });
}

function AddCustomer() {
    $.ajax({
        url: 'http://localhost/Web ApiService/api/Customer',
        type: 'post',
        datatype: 'json',
        data: $('#frmCustomer').serialize(),
        success: ClearAndLoad,
        error: DisplayError
    });
}

function UpdateCustomer() {
    $.ajax({
        url: 'http://localhost/Web ApiService/api/Customer',
        type: 'put',
        datatype: 'json',

```

```

        data: $('#frmCustomer').serialize(),
        success: ClearAndLoad,
        error: DisplayError
    });
}

function DeleteCustomer() {
    var Status = confirm('Are you sure of deleting the current record?');
    if (Status) {
        $.ajax({
            url: 'http://localhost/Web ApiService/api/Customer/' + $('#Custid').val(),
            type: 'delete',
            datatype: 'json',
            success: ClearAndLoad,
            error: DisplayError
        });
    }
}

function ClearAndLoad() {
    $('#frmCustomer').trigger("reset");
    GetCustomers();
    $('#Custid').focus();
}

```

Now write the below code under “<body>” tag by deleting the existing “<div>” tag over there:

```

<body onload="GetCustomers()">


| <table align="center" border="1" id="tblCustomers"> <caption>Customer Details</caption> <thead> <tr> <th>Custid</th><th>Name</th><th>Balance</th><th>City</th><th>Status</th></tr> </thead> <tbody> </tbody> </table> | Custid | Name    | Balance | City   | Status | <form id="frmCustomer"> <table align="center"> <tr> <td>Custid:</td> <td></td> </tr> </table> | Custid: |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|---------|---------|--------|--------|-----------------------------------------------------------------------------------------------|---------|--|
| Custid                                                                                                                                                                                                                | Name   | Balance | City    | Status |        |                                                                                               |         |  |
| Custid:                                                                                                                                                                                                               |        |         |         |        |        |                                                                                               |         |  |


```

```

<input id="Custid" name="Custid" />
<input type="button" id="btnSearch" name="btnSearch" value="..." onclick="GetCustomer()" />
</td>
</tr>
<tr>
<td>Name:</td><td><input id="Name" name="Name" /></td>
</tr>
<tr>
<td>Balance:</td><td><input id="Balance" name="Balance" /></td>
</tr>
<tr>
<td>City:</td><td><input id="City" name="City" /></td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="button" id="btnInsert" name="btnInsert" value="Insert" onclick="AddCustomer()" />
<input type="button" id="btnUpdate" name="btnUpdate" value="Update"
       onclick="UpdateCustomer()" />
<input type="button" id="btnDelete" name="btnDelete" value="Delete" onclick="DeleteCustomer()" />
<input type="reset" id="btnReset" name="btnReset" value="Clear" />
</td>
</tr>
</table>
</form>
</td>
</tr>
</table>
</body>

```

Consuming the Web API Service from an MVC Application using JQuery from a different project: to test consuming our “Web Api service” in a new MVC Project, create a new “ASP.NET Web Application” Project naming it as “WebApiConsumer1”, choose Empty Project Template and select the “MVC” Check Box. Install “JQuery” package using Nuget Package Manager. Add a controller in the project naming it as “HomeController” and add a view to the existing Index Action Method without choosing a layout, copy the complete code which we implemented in “Index.cshtml” of previous project i.e., “Web ApiService” and paste it into the current “Index.cshtml” by deleting all existing content.

Run the project but we will not get the output because of “CORS (Cross Origin Resource Sharing)” i.e., by default for security reasons, browsers restrict Cross-Origin HTTP Requests initiated thru Scripts (JQuery, React, Angular, etc), and to resolve this problem we need to enable “CORS” policy in our “Web ApiService” project so that it lets a Web Application running at one origin (domain) to access resources from an application at a different origin (domain).

To enable CORS in our “Web ApiService” project we need to first install a “Nuget Package” i.e., “Microsoft.AspNet.WebApi.Cors”, so open the Nuget Package Manager and install the package. After installing the packing, we can enable CORS either for the whole Project or to a particular “Web Api” Controller class also.

To enable CORS for a particular Web API Controller class, do the following:

Step: 1 To enable CORS for a particular **Api Controller** class go to **CustomerController.cs** file and add **[EnableCors]** attribute to **CustomerController** class by importing the namespace **"System.Web.Http.Cors"** as following:

```
[EnableCors("*", "*", "*")]
public class CustomerController : ApiController
```

EnableCors attribute expects 3 parameters and those are:

- Origins, which should be a comma separated list of origins that are allowed to access the resource or use "*" to allow all.
- Headers, which should be a comma separated list of headers that are supported by the resource or use "*" to allow all or use "null" or empty string to allow none.
- Methods, which should be a comma separated list of methods that are supported by the resource or use "*" to allow all or use "null" or empty string to allow none.

Step 2: Now go to **"WebApiConfig.cs"** file which is present under **"App_Start"** folder and write the following code in the end of **Register** method which is present inside the class **"WebApiConfig"**:

```
config.EnableCors();
```

To enable CORS for the whole project do the following: to enable CORS for the whole project, without adding **"[EnableCors]"** attribute to each **Web Api Controller** class, directly go to **"WebApiConfig.cs"** file which is present under **"App_Start"** folder and write the below code in the end of **Register** method which is present inside of the class **"WebApiConfig"** by importing the namespace **"System.Web.Http.Cors"**:

```
EnableCorsAttribute obj = new EnableCorsAttribute("*", "*", "*");
config.EnableCors(obj);
```

Note: after enabling **Cors** either **Controller Level** or **Project Level** we can start accessing the **Web API** from other projects also.

Developing a new Web Api Service class with Image loading:

Step 1: Create a **Table** under our **MVCDB Database** with the name **Students** as following:

```
Create Table Students (Id Int Primary Key, Name Varchar(50), Photo VarBinary(max), Status Bit Not Null Default 1);
```

Step 2: Open the **"TestEF.edmx"** file under our **"Web ApiService"** project, right click on it and choose the option **"Update Model from Database"** and choose the **"Students"** table which we have created right now, from the **"Update Wizard"** and click on **Ok Button**, which will add the **"Students"** table as an entity, with the name **"Student"** beside the **"Customer"** entity which we have added earlier.

Step 3: Now add a **"Web API 2 Controller"** under the **Controllers** folder naming it as **"StudentController"** and write the below code in the class:

```
using System.Data.Entity;
using Web ApiService.Models;
public class StudentController : ApiController
{
    MVCDBEntities dc = new MVCDBEntities();
```

```

public List<Student> Get()
{
    return dc.Students.Where(S => S.Status == true).ToList();
}
public Student Get(int Id)
{
    return dc.Students.Find(Id);
}
public HttpResponseMessage Post(Student student)
{
    try
    {
        student.Status = true;
        dc.Students.Add(student);
        dc.SaveChanges();
        return new HttpResponseMessage(HttpStatusCode.Created);
    }
    catch(Exception)
    {
        throw new HttpResponseException(HttpStatusCode.InternalServerError);
    }
}
public HttpResponseMessage Put(Student student)
{
    try
    {
        Student obj = dc.Students.Find(student.Id);
        if (obj == null)
        {
            return new HttpResponseMessage(HttpStatusCode.NotFound);
        }
        else
        {
            obj.Name = student.Name;
            obj.Photo = student.Photo;
            obj.Status = true;
            dc.Entry(obj).State = EntityState.Modified;
            dc.SaveChanges();
            return new HttpResponseMessage(HttpStatusCode.OK);
        }
    }
    catch (Exception)
    {
        throw new HttpResponseException(HttpStatusCode.InternalServerError);
    }
}

```

```

public HttpResponseMessage Delete(int Id)
{
    try
    {
        Student obj = dc.Students.Find(Id);
        if (obj == null)
        {
            return new HttpResponseMessage(HttpStatusCode.NotFound);
        }
        else
        {
            obj.Status = false;
            dc.Entry(obj).State = EntityState.Modified;
            dc.SaveChanges();
            return new HttpResponseMessage(HttpStatusCode.OK);
        }
    }
    catch (Exception)
    {
        throw new HttpResponseException(HttpStatusCode.InternalServerError);
    }
}
}

```

Note: build the project and compile it, so that we can consume it from other projects.

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:

```

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public byte[] Photo { get; set; }
}

```

The current **MVC Application** is being designed to consume a **Web API Service** and to refer the service we need the URL of the Service. So, go to “**Web.config**” file and add the following element under “**<appSettings>**” tag:

```
<add key="WebApiUrl" value="http://localhost/Web ApiService/Api/" />
```

Add a controller in the Controllers folder naming it as “**StudentController**” and write the below code in it deleting the existing **Index** action method:

```
using System.IO;
using System.Net.Http;
using System.Configuration;
using System.Threading.Tasks;
using WebApiConsumer2.Models;
public class StudentController : Controller
{
    HttpClient client = new HttpClient();
    string serviceUrl = ConfigurationManager.AppSettings.Get("WebApiUrl");
    public ViewResult DisplayStudents()
    {
        client.BaseAddress = new Uri(serviceUrl);
        Task<HttpResponseMessage> getTask = client.GetAsync("Student");
        getTask.Wait();
        HttpResponseMessage message = getTask.Result;
        List<Student> students = message.Content.ReadAsAsync<List<Student>>().Result;
        return View(students);
    }
    public ViewResult DisplayStudent(int Id)
    {
        client.BaseAddress = new Uri(serviceUrl);
        Task<HttpResponseMessage> getTask = client.GetAsync("Student/" + Id);
        getTask.Wait();
        HttpResponseMessage message = getTask.Result;
        Student student = message.Content.ReadAsAsync<Student>().Result;
        return View(student);
    }
    public ViewResult AddStudent()
    {
        return View(new Student());
    }
    [HttpPost]
    public ActionResult AddStudent(Student student, HttpPostedFileBase selectedFile)
    {
        if (selectedFile != null)
        {
            BinaryReader br = new BinaryReader(selectedFile.InputStream);
            student.Photo = br.ReadBytes(selectedFile.ContentLength);
        }
    }
}
```

```

client.BaseAddress = new Uri(serviceUrl);
Task<HttpResponseMessage> postTask = client.PostAsJsonAsync("Student", student);
postTask.Wait();
HttpResponseMessage message = postTask.Result;
if (message.IsSuccessStatusCode)
{
    return RedirectToAction("DisplayStudents");
}
else
{
    return View();
}
}

public ViewResult EditStudent(int Id)
{
    client.BaseAddress = new Uri(serviceUrl);
    Task<HttpResponseMessage> getTask = client.GetAsync("Student/" + Id);
    getTask.Wait();
    HttpResponseMessage message = getTask.Result;
    Student student = message.Content.ReadAsAsync<Student>().Result;
    if (student.Photo != null)
    {
        Session["Photo"] = student.Photo;
    }
    return View(student);
}

public ActionResult UpdateStudent(Student student, HttpPostedFileBase selectedFile)
{
    if (selectedFile != null)
    {
        BinaryReader br = new BinaryReader(selectedFile.InputStream);
        student.Photo = br.ReadBytes(selectedFile.ContentLength);
    }
    else if (Session["Photo"] != null)
    {
        student.Photo = (byte[])Session["Photo"];
    }
    client.BaseAddress = new Uri(serviceUrl);
    Task<HttpResponseMessage> putTask = client.PutAsJsonAsync<Student>("Student", student);
    putTask.Wait();
    HttpResponseMessage message = putTask.Result;
    if (message.IsSuccessStatusCode)
    {
        return RedirectToAction("DisplayStudents");
    }
}

```

```

    {
        return View("EditStudent", student);
    }
}

public ActionResult DeleteStudent(int Id)
{
    client.BaseAddress = new Uri(serviceUrl);
    Task<HttpResponseMessage> deleteTask = client.DeleteAsync("Student/" + Id);
    deleteTask.Wait();
    HttpResponseMessage message = deleteTask.Result;
    if (message.IsSuccessStatusCode)
    {
        return RedirectToAction("DisplayStudents");
    }
    else
    {
        return View();
    }
}
}

```

Add a view with the name **DisplayStudents.cshtml**, selecting layout **Checkbox** and write the below code in it by deleting the whole content in the View:

```

@model IEnumerable<WebApiConsumer2.Models.Student>
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
<table border="1" align="center">
    <caption>Student Details</caption>
    <tr>
        <th>@Html.DisplayNameFor(S => S.Id)</th>
        <th>@Html.DisplayNameFor(S => S.Name)</th>
        <th>@Html.DisplayNameFor(S => S.Photo)</th>
        <th>Actions</th>
    </tr>
    @foreach (var Student in Model)
    {
        <tr>
            <td>@Html.DisplayFor(S => Student.Id)</td>
            <td>@Html.DisplayFor(S => Student.Name)</td>
            @{
                string imgSrc = "";
                if (Student.Photo != null)
                {
                    var base64 = Convert.ToBase64String(Student.Photo);
                    imgSrc = String.Format("data:image/jpeg;base64,{0}", base64);
                }
            }
        </tr>
    }

```

```

        }
    <td><img src='@imgSrc' width="40" height="25" /></td>
    <td>
        @Html.ActionLink("View", "DisplayStudent", new { Id = Student.Id }) &nbsp;
        @Html.ActionLink("Edit", "EditStudent", new { Id = Student.Id }) &nbsp;
        @Html.ActionLink("Delete", "DeleteStudent", new { Id = Student.Id },
            new { onclick = "return confirm('Are you sure of deleting the record?') })
    </td>
</tr>
}
<tr>
    <td colspan="4" align="center">@Html.ActionLink("Add New Student", "AddStudent")</td>
</tr>
</table>

```

Generate a view for `DisplayStudent` action method and while adding the View, choose the **Template as Empty**, **Model Class** as `Student`, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “`<h2>`” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
{@
    string imgSrc = "";
    if (Model.Photo != null) {
        var base64 = Convert.ToBase64String(Model.Photo);
        imgSrc = String.Format("data:image/jpeg;base64,{0}", base64);
    }
}
<table border="1" align="center">
    <caption>Student Details</caption>
    <tr>
        <td rowspan="2"><img src='@imgSrc' width="200" height="200" /></td>
        <td>Sid: @Model.Id</td>
    </tr>
    <tr>
        <td>Name: @Model.Name</td>
    </tr>
    <tr>
        <td colspan="2" align="center">@Html.ActionLink("Back to Student Details", "DisplayStudents")</td>
    </tr>
</table>

```

Generate a view for `AddStudent` action method and while adding the View, choose the **Template as Empty**, **Model Class** as `Student`, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “`<h2>`” element present in it:

```
<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Add New Student</h2>
```

```

@using (Html.BeginForm("AddStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Id)<br />@Html.TextBoxFor(S => S.Id)</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>@Html.LabelFor(S => S.Photo)<br /><input type="file" name="selectedFile" /></div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
        <input type="reset" value="Reset" name="btnReset" />
    </div>
}
@Html.ActionLink("Back to Student Details", "DisplayStudents")

```

Generate a view for **EditStudent** action method and while adding the View, choose the **Template as Empty**, **Model Class** as **Student**, choose “**Use a layout page**” CheckBox in the “**Add View**” window and write the below code over there by deleting the existing “**<h2>**” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Edit Student</h2>
{@
    string imgSrc = "";
    if (Model.Photo != null)
    {
        var base64 = Convert.ToBase64String(Model.Photo);
        imgSrc = String.Format("data:image/jpeg;base64,{0}", base64);
    }
}
@using (Html.BeginForm("UpdateStudent", "Student", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div>@Html.LabelFor(S => S.Id)<br />@Html.TextBoxFor(S => S.Id, new { @readonly = "true" })</div>
    <div>@Html.LabelFor(S => S.Name)<br />@Html.TextBoxFor(S => S.Name)</div>
    <div>
        @Html.LabelFor(S => S.Photo) <br /><img src='@imgSrc' width="200" height="200" />
        <input type="file" name="selectedFile" />
    </div>
    <div>
        <input type="submit" value="Save" name="btnSave" />
    </div>
}

```

Add a view for **DeleteStudent** Action method choosing a layout and write the below code in it by deleting the existing “**<h2>**” element present in it:

```

<h2 style="text-align:center;background-color:yellowgreen;color:orangered">Delete Student Failed</h2>
<font color="red" size="4">
    Delete student action resulted in an error.
</font>
@Html.ActionLink("Back to Student Details", "DisplayStudents")

```

ASP.NET Core

Nowadays, when it comes to the software development, everyone is talking about **free**, **open-source**, and **cross-platform** development. As we all know **Microsoft** is well known for its **Windows-Based** products. Now we are in the new age of software development and for this, a new revolutionary product came into the market from **Microsoft**, and it is **ASP.NET Core**.

History of ASP.NET: As we know, **ASP.NET** is a framework that has been used to develop data-driven **Web Applications** for many years. Since then, **ASP.NET Framework** went through a steady evolutionary change, and finally, the most decent evolution is **ASP.NET Core**.

ASP.NET Core is not a continuous part of the **ASP.NET 4.x Framework**, but instead it is a completely new **Framework**. This **Framework** is actual a re-write of the current **ASP.NET 4.x Framework**, but with much smaller and a lot more modular. Some people think that many things remain the same, but that is not completely true. The **ASP.NET Core** is a big fundamental change to the **ASP.NET Framework**.

What is ASP.NET Core?

Ans: **ASP.NET Core** is a new version of **ASP.NET Web Framework** mainly targeted to run on **.NET Core**. It is **free**, **open-source**, **high-performance**, **lightweight**, and **cross-platform** framework for building **cloud-based** applications, such as **Web Apps**, **IoT Apps** and **Mobile Apps**. It is designed to run on the **cloud** as well as **on-premises**. Same as **.NET Core** it was architected **modular** with **minimum overhead** and then other more **advanced features** can be added as **Nuget Packages** as per application requirements. This result in **high performance**, **require less memory**, **less deployment size**, and **easy to maintain**.

ASP.NET Core 8.0, 7.0, 6.0, 5.0 and 3.x application can be targeted to develop and run-on top of the **.NET Core** only whereas **ASP.NET Core 1.x and 2.x** application can be targeted to develop and run-on top of the **.NET Core (Cross-platform)** as well as **.NET Framework (Windows only)**.

Note: **ASP.NET Core** was initially launched as **ASP.NET 5** but later it's renamed to **ASP.NET Core**.

Why ASP.NET Core?

Ans: Nowadays, the **ASP.NET Core Framework** became more and more popular among developers. There are **several reasons** why **modern developers** are using it which are listed below:

Open Source: **ASP.NET Core Framework** is **Open Source**, and this is the main reason behind its **popularity**. The entire source code of this **ASP.NET Core** is available at => <https://github.com/aspnet> and you are free to download the source code and even you can also modify and compile your own version of it. **".NET Core"** team is always there to support your effort in the **seamless development** of the application.

Cross-Platform: The **ASP.NET Core Framework** is designed from scratch to keep in mind to be **Cross-Platform** for both **development** and **deployment** also. So, we don't need to build **different applications** for **different platforms** using **different Framework's**. The **earlier version** of **ASP.NET Framework** applications can run only on **Windows Platforms**, whereas **ASP.NET Core** applications can be **developed** as well as **run on different platforms** such as **Windows, Mac, or Linux**. We can **host** the earlier **ASP.NET Framework** applications on **IIS** only whereas we can **host** the **ASP.NET Core** applications on **IIS, Nginx, Docker, Apache**, or even **self-host** deployment.

Editors: To develop **ASP.NET Core** applications we have multiple options like **Visual Studio** or **Visual Studio Code** as well as if required, we can also use any **third-party editors** as per your choice.

CLI Support: Using **CLI (Command Line Interface) commands** you can develop and run **.NET applications** as well as you can also **publish** the application using **CLI command**.

Fast: **ASP.NET Core** no longer depends on “**System.Web.dll**” for **browser-server** communication. **ASP.NET Core** allows us to **include packages** that we need for the application and this will **reduce the request pipelines** and improves performance and **scalability**.

IoC Container: One of the most important used design patterns in the real-time application is the “**Dependency Injection-Design Pattern**”. It includes the built-in **IoC (Inversion of Control)** container for automatic dependency injection which makes it maintainable and testable.

Unified MVC and Web API Framework: The **ASP.NET Core** provides a unified programming model for developing both **Web Apps** and **Web API**. That means a single controller class can be used to handle both. A **Controller** we create in **ASP.NET Core** (either **Web App's** or **Web API**) application is going to inherit from the **Controller** base class and all their action methods returns the “**IActionResult**” interface. The “**IActionResult**” interface provides several implementations like “**JsonResult**”, “**ViewResult**” etc.

Integration with Modern UI Framework: It allows you to use and manage modern **UI Frameworks** such as **Angular JS**, **Angular**, **React JS**, and **Bootstrap**, etc.

Handling Request and Response Pipeline: We can handle the **request** and **response** in the **ASP.NET Core** application by using new **Middleware Components**. In earlier **ASP.NET 4.x** we generally use **Handlers** and **Modules** to handle the **Request** and **Response** pipeline. The **ASP.NET Core Framework** provides lot of built-in **Middleware Components** and we can use those **Components** to handle the **request** and **response** pipeline. If we want, we also can also create our own **Middleware** components and use them.

Hosting: **ASP.NET Core Web App's** can be hosted on **multiple platforms** with-in any **Web Server** such as **IIS**, **Apache**, **Nginx**, **Docker**, etc. It is not dependent only on **IIS** as our standard **.NET Framework Web App's**.

Side-by-Side Application Versioning: **ASP.NET Core** runs in **.NET Core**, which supports the simultaneous running of **multiple versions** of applications.

Smaller Deployment Footprint: **ASP.NET Core** application runs on **.NET Core**, which is smaller than the full **.NET Framework**. So, application which uses “**.NET Core FX**” libraries will be having a **smaller deployment size**, and this reduces the **deployment footprint**.

Note: Many people are confused between “**ASP.NET Core**” and “**.NET Core**”. Please note that **ASP.NET Core** and **.NET Core** are not the same. They are different, just like **ASP.NET** and **.NET Framework** are different. To understand, **.NET Core** is a **fast**, **lightweight**, **modular**, and **open-source** framework for creating **Web Applications**, **Desktop Applications** (**CUI**, **GUI**), **Mobile Applications**, **Micro services**, **Cloud**, **Machine Learning**, **Game Development**, and **Internet of Things** (**IOT**) that can run on **Windows**, **Linux**, and **Mac OS**, whereas **ASP.NET Core** is a **Web Application Framework** under **.NET Core** for building **Web Applications** only. So, **.NET Core** is a **Software Platform** on which **ASP.NET Core Applications** run.

.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 available for ASP.NET Core. .NET Core runtime and SDK includes ASP.NET Core libraries also.
.NET Core 8.0 - latest version.	ASP.NET Core 8.0 - 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?

Ans: 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
- Packages.config file
- HTTP Handlers and HTTP Modules

Creating ASP.NET Core Web Application

To create an **ASP.NET Core Web Application**, open **Visual Studio 2022**; click on the **Create a new project** button which will open “Create a new project” window. This window displays different **.NET Core** application templates, choose “**ASP.NET Core Empty**” template and click on the **Next** button which will open “**Configure your new project**” window and here we need to give a name for our project for example “**CoreTestProject1.Core5.0**”, and specify the location to save and click on the Next button, which will open “**Additional Information**” window and here we need to select the “**Target Framework**” to build the application choose “**.NET 5.0 (Out of support)**” and also uncheck all the checkboxes on this window and click on **Create** button. Once we click on the **Create** button, it will create a new “**ASP.NET Core Empty Project**” and here we can understand how different components fit together to develop an “**ASP.NET Core application**”. The project will be created with the following files and folder structure:

CoreTestProject1.Core5.0

- Connected Services
- Dependencies
- Properties
 - ❖ launchSettings.json
- appsettings.json
 - ❖ appsettings.Development.json
- Program.cs
- Startup.cs

Note: Until **ASP.NET Core 5.0** we have a different coding style and from **ASP.NET Core 6.0** we have a different coding style. To understand the differences let us create a project in latest version of **ASP.NET Core** i.e., **8.0**.

Now same as the above project, create one more project with the name “**CoreTestProject1.Core8.0**” and here select the “**Target Framework**” as “**.NET 8.0 (Long-term support)**”, un-check “**Configure for HTTPS**” and “**Enable Docker**” Checkbox’s, and check “**Do not use top-level statements**” Checkbox on this window, and click on **Create** button. This project will be created with the following files and folder structure:

CoreTestProject1.Core8.0

- Connected Services
- Dependencies
- Properties
 - ❖ launchSettings.json
- appsettings.json
 - ❖ appsettings.Development.json
- Program.cs

Note: Up to **ASP.NET Core 5.0** we find the file “**Startup.cs**” whereas from **ASP.NET Core 6.0** we will not have “**Startup.cs**” file anymore.

Run the **ASP.NET Core Application’s** which will open a browser and displays the output as “**Hello World!**” and this output comes from the **Configure** method of the **Startup** class which is present inside the “**Startup.cs**” file in case of **ASP.NET Core 5.0**, open the file and watch the code present in the **Configure** method and there you will find the below statement in end of the method:

```
await context.Response.WriteAsync("Hello World!");
```

If we are working with **ASP.NET Core 6.0 and above** then this output comes from “**Program.cs**” file, open the file and watch the code present over there, which displays a statement as below:

```
app.MapGet("/", () => "Hello World!");
```

Change the “**Hello World!**” string to something else and re-run the projects again which will display the output accordingly. For example, let’s change the **WriteAsync** and **MapGet** methods code as following:

```
5.0 => await context.Response.WriteAsync(".NET Core Empty Web Application created using ASP.NET Core 5.0");
```

```
8.0 => app.MapGet("/", () => ".NET Core Empty Web Application created using ASP.NET Core 8.0");
```

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

The **ASP.NET Core Web App (Model-View-Controller)** template contains everything that is required to create an **ASP.NET Core MVC Web Application** and this creates **Models**, **Views**, and **Controller’s** folders by default. It also adds web-specific things such as **Java Script**, **CSS files**, **Layout files**, etc. which are necessary and required to develop an **MVC Application**.

To create an **MVC Application**, open **Visual Studio**, click on create a new project button, in the window opened choose “**ASP.NET Core Web App (Model-View-Controller)**” template and click on **Next** button, in the window opened provide a name to the project as “**CoreMVCProject5**”, specify the location to save and click on **Next** button, in the window opened choose **Target Framework** as “**.NET 5.0 (Out of support)**” and also make sure the **Checkbox “Configure for HTTPS is selected”**, and click on the **Create** button which creates the project with the following files and folder structure:

CoreMVCProject5

- Connected Services
- Dependencies

- Properties
 - ❖ launchSettings.json
- wwwroot
 - ❖ css
 - ❖ js
 - ❖ lib
- Controllers
- Models
- Views
- appsettings.json
 - ❖ appsettings.Development.json
- Program.cs
- Startup.cs

Connected Services: This is the first element in our project structure and possibly the less used one. It is intended to automate the steps necessary to connect a project to an external service (like **Azure Storage**).

Dependencies: This element of your project structure contains information of all **packages** and **projects** on which our project depends. There are four main parts inside this node:

1. **Analyzers:** They help you make your code better, cleaner, and error-free. Each analyzer checks that your code satisfies a list of rules incorporated in it. If any part of your code does not apply to one of the rules, you will see either **warning** or an **Error** while you build your project. Please note that analyzers work only at compile time and do not affect your resulting application.
2. **Frameworks:** This contains a list of frameworks your project depends on. This information is important if you publish your web application to a **Server** (as opposed to a **self-contained** one). In this case, all the frameworks listed here must be installed on the **Server** where you will run your app.
3. **Packages:** This is the main item which lists all the **Nuget** packages you added to your project. If any of those packages depend on other packages, they will be installed automatically and listed as sub-nodes of the root-level packages. You can **remove** each **installed** package here (**right-click => Remove**).
4. **Projects:** This is the list of other projects our project depends on in the current solution. You can reference other projects by using the “**Add Project Reference**” command from the right-click menu.

Properties: This contains different **properties** of our project that we can modify by double-clicking on this node in the **Solution Explorer**. Most of the properties here, affect the **compile-time** and **debug-time** behavior of your project. The only item inside of this node is “**launchSettings.json**” file that contains the list of the **launch profiles**. Each **profile** defines how to run your project, whether the browser should be opened or not, which port to use, which environment variables should be set when we “**Run**” the project, etc. By default, we will be finding 2 profiles to run the **ASP.NET Core Web Application** and those are: “**IIS Express**” profile and “**CoreMVCProject5**” profile. The “**launchSettings.json**” file is as below in the project we have created:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
```

```

    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:53245",
      "sslPort": 44375
    },
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CoreMVCProject5": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

Note: We can edit “`launchSettings.json`” file by using the project properties window also and to do that double click on the Properties node in **Solution Explorer** and in the window opened, click “**Debug Tab**” in **LHS** and this provides the option for editing “**Launch Settings**” and to do that click on “**Open debug launch profiles UI**”, link.

wwwroot: This folder is known as a “**web root**” folder and contains all the **static files** of our web application like **CSS files**, **Java Script files**, **Html files**, and **Image files**. As you might figure out from its name, this will be the **root folder** of our **web application**. So, if we add a new folder under “**wwwroot**” with the name “**images**” and under the folder if we add an image file with the name “**Autumn.jpg**”, then we can access that image from browser by using the address: “**/images/Autumn.jpg**”. To test this run the application hitting “**F5**” and then append “**images/Autumn.jpg**” to the “**URL**” in the address which should now look as following in case of IIS Express or Kestrel Web Servers: <https://localhost:<port>/images/Autumn.jpg>

Note: In traditional **ASP.NET Application** (i.e., **Framework**), static files can be served from the root folder of an application or any other folder under it, which has been changed in **ASP.NET Core**. Now those files that are present in the web root - “**wwwroot**” folder can be served over an **http request**. All other files are blocked and can't be served by default. Generally, there will be separate folders for different types of static files such as **Java Script**, **CSS**, **Images**, and **Library Scripts** etc. in the “**wwwroot**” folder.

We can rename “**wwwroot**” folder to any other name as per our choice and set it as a web root while preparing hosting environment in “**Program.cs**”. For example, if we change the “**wwwroot**” folder name as “**myroot**” we need to specify that in **Program** class under “**CreateHostBuilder**” method.

By default, code in the method will be as following:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
```

```

Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.UseStartup<Startup>();
});

```

Change the above code as below to register our myroot folder as web root folder:

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.UseStartup<Startup>().UseWebRoot("myroot");
});

```

Controllers: this folder contains all the **Controller** classes we define in our project.

Models: this folder contains **Model** classes i.e., **Classes** representing the **Entities** and **Properties** representing the **Attributes** of **Entities** as well as all the **Methods** to manipulate the data.

Views: this folder contains all the **View** files that are required for this application and the extension of these files will be “**.cshtml**” and we call these files as “**Razor Pages**”.

Till now we have looked over all the main folder's that are present with-in the “**ASP.NET Core MVC Application**” project, now let's take a closer look at the files stored in the project's root folder which are also most important part of the project.

appsettings.json: this file is used to store information such as **connection strings** or application specific settings and they are stored in **JSON** format, as the file extension suggests. If you are familiar with **ASP.NET** you may notice that the function of this file is like “**Web.config**” file, and right now we find the below code in it:

```

{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*"
}

```

The settings in the file have a **hierarchical** structure and can be accessed with the “**Configuration**” object defined in the **Startup** class or in any other place of your program where you injected “**IConfiguration**” service. For example, if we want to read the value of key “**Default**” or “**Microsoft**” then the code will be as following:

```

var Data1 = Configuration.GetValue<string>("Logging:LogLevel:Default");
var Data2 = Configuration.GetValue<string>("Logging:LogLevel:Microsoft");

```

By default, in addition to “**appsettings.json**” file, the **ASP.NET Core Project** also includes an “**appsettings.Development.json**” file and all the settings defined in this file are loaded only in the **Development**

Environment and are not available in **Production**. Same as this we can also define “`appsettings.Production.json`” for settings in **Production Environment** and “`appsettings.Staging.json`” for settings in **UAT Environment**.

Program.cs: This file contains a class in it with the name **Program** and a static “**Main**” method which is the entry point of our “**Application**”. Just like our **Console App’s** and **Windows App’s**, **ASP.NET Core App’s** also starts from **Program** class only. The purpose of this method is to define the “**host**” and then pass the control to the “**Startup**” class.

Code in Program class will be as following by default:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

A host is an object that encapsulates an app’s resources, such as: **Logging**, **Dependency Injection (DI)**, **Configuration**, **IHostedService** Implementations, etc. As shown in the above code, `CreateHostBuilder()` method returns an object that implements the **IHostBuilder** interface. Host is a static class that can be used for creating an instance of **IHostBuilder** with pre-configured defaults by calling `CreateDefaultBuilder()` method which will create a new instance of **HostBuilder** with pre-configured defaults.

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**, which 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 MVC 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;
}
```

- The `ConfigureServices` method, where we register all necessary services like “Authentication/Authorization Service”, “MVC or Razor Page or Web Api 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 “`services.AddControllers`”, etc. All these methods are extension methods.

- 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.

Note: 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.

By default, in an ASP.NET Core MVC Application the method contains below code in it:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see
           https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseStaticFiles();

    app.UseRouting();
```

```

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
}

```

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 and Staging Mode**, we catch all exceptions and then re-direct the request to the specified path i.e., "**Home/Error**" in our case.

Processing static files (UseStaticFiles) Middleware: the next middleware in the pipeline is "**UseStaticFiles()**" which takes care of all static files, i.e., if the application gets a request for ".js", ".css", or an image files, this middleware looks for a file with the requested name inside of the "**wwwroot**" folder and returns it back if found. If not found it will send back a response with a **404-status code** (not found).

Routing (UseRouting and UseEndpoints): the next pair of middleware's are, the most important ones in the pipeline since they define the routing for all other endpoints in your web application. In simple words, they match a particular request to a particular endpoint, a piece of executable code that handles the request.

How exactly does it work?

Ans: In the **Configure** method first we call "**app.UseRouting()**" middleware to add the "**Endpoint Routing**" middleware to our pipeline and after that, we need to call "**app.UseEndpoints()**" middleware to add "**Endpoint**" middleware to the pipeline and define the endpoints. Each endpoint is an object that contains (as mentioned above) a delegate (so, a piece of code) that handles the request. We can use such extension methods as **MapGet**, **MapPost**, and others to add an endpoint that matches a particular request path or a path template. For example, the following pattern "**/something/{path*}**", will be matched for all requests started with "**/something/**". To test this, change the pattern in the code as following:

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

Now all the requests should contain the word "NIT" before the controller's name, as following:

http://localhost:port/NIT/Controller_Name/Action_Name => <http://localhost:port/NIT/Home/Index>

When the web application gets a new request, it's not processed by any middleware defined before "**UseRouting**" middleware and then the **Endpoint Routing Middleware** matches it to some **endpoint**. Then **Endpoint Middleware** calls the **endpoint's delegate** to handle the request. All other **middleware's** that are added after **app.UseRouting()** and before **app.UseEndpoints()** can see which endpoint was selected by **EndpointRouting Middleware** and they can change something (e.g., apply an **authorization** policy) before **EndpointMiddleware** dispatches to the selected **endpoint**.

Authorization (UseAuthorization): this middleware is added for **authorizing** a user if the current request is **anonymous**, but the selected endpoint requires authorization.

Both “ConfigureServices” and “Configure” methods are called **implicitly** while the application starts. You just only need to define the **services** and **middleware’s** here correspondingly.

Note: If the same project is created using **.NET 6.0 and above Framework** then we will not find the class “**Startup**” and the code of this class will be present directly under “**Program**” class only. By default, the code under “**Program.cs**” file will be with-out any **class** over there, because **.NET 6.0** uses **C# 10.0** version and in this version, we are provided with a feature called **Top-Level Statement’s** i.e., if the project has a single **Main** method and all the code, if required to be defined under the **Main** method only then with-out explicitly defining a **Class** and **Main** method it will directly write the code in the file as below:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Note: if you still want to see the class; while creating the project, in the window where we select the **Framework** version, select the Checkbox **“Do not use top-level statements”** so that you will find the code with a **class** and **Main** method also. To test that create a new **MVC Application** project naming it as “**CoreMVCProject8**”, and in the window where we choose **Target Framework** select “**.NET 8.0 (Long-term support)**” and select the Checkbox’s **“Configure for HTTPS”** and **“Do not use top-level statement’s”**, click on **Create** button which creates the project with-out a **Startup** class and code under the **Program** class will be as following:

```
namespace CoreMVCProject8
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");

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

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
}
}
}

```

Note: Because we don't have **Startup** class here and all code need to be defined in the **Main** method of **Program** class only, follow the below guidelines to write the code:

- All the code we need to write under "**ConfigureServices**" method of "**Startup**" class should be implemented before this statement => "`var app = builder.Build();`".
- All the code we need to write under "**Configure**" method of "**Startup**" class should be implemented after this statement => "`var app = builder.Build();`".

Why do we have a Main method in ASP.NET Core?

Ans: The most important point that you need to keep in mind is, an **ASP.NET Core Web Application** initially starts as a "**Console Application**" and the **Main** method is an **entry point** to that application. When we execute the **ASP.NET Core Web Application** first it looks for **Main** method and this is the method from where the execution starts. **Main** method will then configure **ASP.NET Core** and starts it, and at this point of time the application becomes a **Web Application**. If we look at the body of **Main** method, there we will find that, it makes a call to "**CreateHostBuilder()**" method by passing the command line arguments "**args**" as a parameter value and the code in the "**Main**" method will be as following:

```

public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

```

Within the `Main` method, on this “`IHostBuilder`” object, the `Build` method is called which builds a `Web Host`. Then it hosts our `ASP.NET Core Web Application` within that `Web Host (Web Server)`. Finally, on the `Web Host (Web Server)`, it calls the `Run` method, which will run the `Web Application` and it starts listening to the incoming `HTTP Requests`.

`CreateHostBuilder` method returns an object that implements `IHostBuilder` interface. `Host` is a static class that can be used for creating an instance of `IHostBuilder` with pre-configured defaults by calling its `CreateDefaultBuilder` method which will create a new instance of `HostBuilder` with pre-configured defaults. Internally, it will also configure `Kestrel` (Internal Web Server for ASP.NET Core), `IIS Integration`, and other configurations. Below is the code we find in `CreateHostBuilder` method:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

As part of setting the Web Host, the `CreateDefaultBuilder` method will do several things, like:

- Setting up the Web Server.
- Loading the application configuration from various configuration sources.
- Configuring logging.

Configuring and setting up the Web Server by `CreateDefaultBuilder` method:

`ASP.NET Core Web Application` can be hosted in “`IIS Express`” or “`IIS`” or “`Kestrel`” Web Servers and supports 2 different hosting models, those are:

- In Process Hosting
- Out of Process Hosting

Note: when we create a new `ASP.NET Core MVC Application` by default it is created with `In-Process` hosting model for hosting the application in `IIS` or `IIS Express` or `Kestrel`. To verify that, open `Project Properties Window` and in that on `LHS` select the option “`Debug`” and on the right, click on “`Open debug launch profiles UI`” link which opens a window and, in that select, “`IIS Express`” on `LHS` and on the right we find “`Hosting Model`” option with the default value “`In Process`”.

In case of `In-Process` hosting, “`CreateDefaultBuilder`” method when sees the value as `In-Process`, will internally host the application inside “`IIS Worker Process`” i.e., “`iisexpress.exe`” for “`IIS Express`”, “`w3wp.exe`” for “`IIS`” and “`<ProjectName>.exe`” in case of “`Kestrel`”. By default, `Visual Studio` uses `IIS Express` to run `Web Applications` up to `ASP.NET Core 5.0` and `Kestrel` to run `Web Applications` from `ASP.NET Core 6.0`.

What is IIS Express?

Ans: `IIS Express` is a lightweight, self-contained version of `IIS` which is designed for `Web Application Development`. The most important point that you need to remember is we use `IIS Express` only in `development`, not on `production` or `staging` and in `production` and `staging` we generally use `IIS`.

What is IIS Web Server, and how to run `ASP.NET Core Web Application` in IIS Web Server?

Ans: Internet Information Services (`IIS`) is a flexible, secure, and manageable `Web Server` for hosting anything on the `Web`. To host our `ASP.NET Core Web Application` on `IIS Web Server` first open `Visual Studio` in `Administrator Mode`,

go to **Project Properties** => select **Debug Tab** in the **LHS** and on the right, click on “**Open debug launch profiles UI**” link which opens a window and, in that click on “**Create a new profile**” option in the **LHS-Top** and select the option **IIS** which adds a new **Profile** with the name as “**Profile1**” rename it as “**IIS**” and now on the right fill the below details:

- Under “**Environment Variables**” Textbox enter the value as “**ASPNETCORE_ENVIRONMENT=Development**”.
- Check the “**CheckBox**” Launch Browser.
- Under “**App URL**” Textbox enter the value as: <http://localhost/CoreMVCProject5> for **CoreMVCProject5** and <http://localhost/CoreMVCProject8> for **CoreMVCProject8**.

Leave rest of the options as is and close the window. Now in the “**Standard Toolbar**” under “**Debug Target DropDownList**” by default it will be showing “**IIS Express**” in **ASP.NET Core 5.0** and “**Project Name (Kestrel)**” from **ASP.NET Core 6.0**, change it to “**IIS**”, run the **project** and watch the **URL** in address bar. Now if we open the “**launchSettings.json**” file we will find some changes made to the file as following:

Under “iisSettings” it will add a new “Name-Value” item as following:

```
"iis": {  
  "applicationUrl": "http://localhost/CoreMVCProject",  
  "sslPort": 0  
}
```

In the bottom of the file, it will create a new Profile i.e., “IIS” Profile as following:

```
"IIS": {  
  "commandName": "IIS",  
  "launchBrowser": true,  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

What is Kestrel Web Server and how to run ASP.NET Core Web Application in Kestrel Web Server?

Ans: As we already know that **ASP.NET Core** is a **Cross-Platform Framework**, which means it supports us to develop and run our applications on different **Operating System's** such as **Windows, Linux, or Mac**. The **Kestrel** is the **Cross-Platform Web Server** for **ASP.NET Core App's**, which means this **Web Server** supports all the platforms and versions that **ASP.NET Core** supports. By default, it is included as an **Internal Web Server** in the **.NET Core Application**.

The **Kestrel Web Server** generally used as an edge server i.e., the internet facing **Web Server** which directly processes incoming **HTTP Request** from the clients. In case of **Kestrel Web Server**, the **Process Name** that is used to host and run the **ASP.NET Core Application** is our **Project Name** only and to check this open “**launchSettings.json**” file and we will find the below code:

```
"CoreMVCProject5": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  },
```

```

    "applicationUrl": https://localhost:5001;http://localhost:5000,
    "dotnetRunMessages": "true"
}

```

Under profile settings we find 3 profiles “IIS Express”, “CoreMVCProject5” (our Project Name) and “IIS” (which we have created explicitly). “IIS Express” profile is used to run the application under IIS Express and we find the “iisSettings” on the top of “launchSettings.json” file and the Application URL is: <http://localhost:PortNo> (Port No. will vary from project to project and machine to machine) and below this we also find SSL Port with some value, because we created our project for “https” protocol. “CoreMVCProject5” profile is used to run the application by using the Kestrel Web Server and the Application URL’s are: <http://localhost:5000> and <https://localhost:5001> (if SSL option has been selected by us while creating the project).

In the “Standard Tool Bar” under “Debug Target DropDownList” of Visual Studio, we find “IIS Express” selected by default, so when we hit F5 and run the project it will run using “IIS Express.exe” and in the browsers address bar we see the Port No. which is shown in “launchSettings.json” file. We can change the profile from “IIS Express” to “CoreMVCProject5” so that the application runs on Kestrel Web Server and now we see the Port No. as 5000 for Http Protocol and 5001 forHttps Protocol and notice that it will launch a new Command Window running the Kestrel Web Server on Port No. 5000 and 5001 (Provided Http’s option is selected while project creation).

Note: We can also run the ASP.NET Core Application from the command line also by using the “.NET Core CLI (Command Line Interface)” which will use Kestrel as Web Server. To run .NET Core Application using .NET Core CLI Command, open “Developer Command Prompt for VS”. Now change the directory to the folder where we saved our Project i.e., “<Drive>:\<Personal_Folder>\CoreMVCProject5\CoreMVCProject5”, so change to that folder and execute the “dotnet run” command as shown below:

```
<Drive>:\<Personal_Folder>\CoreMVCProject5\CoreMVCProject5> dotnet run
```

Once we type the “dotnet run” command and press enter key, .NET Core CLI builds and runs the application and it also shows the URL to access our application as following => <http://localhost:5000> and <https://localhost:5001> (provided Https option is selected while project creation), and if you remember these port are configured in “launchSettings.json” file of our application under “CoreMVCProject5” profile which is nothing but the profile for the Kestrel Web Server. Now open any browser and navigate to <http://localhost:5000> or <https://localhost:5001> (provided Http’s option is selected while project creation), which displays the output.

Note: Till now even we ran our application in “IIS Express” or “IIS” or “Kestrel Web Server” all these are “In-Process Hosting” only and now let’s learn about “Out of Process Hosting”.

Out-Of-Process Hosting: to run the application in “OutOfProcess” we need to configure it either in the “Project File” or under “Debug” option of Project Properties Window.

Option 1: Configuring in Project File: in Solution Explorer, right click on the Project and select the option “Edit Project File” which will open “CoreMVCProject5.csproj” file in the document window. In this file we need to add the <AspNetCoreHostingModel> tag under <PropertyGroup> tag with value as “OutOfProcess” as shown below:

```
<AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
```

Option 2: Specify the Hosting Model as OutOfProcess in Project Properties Window: open the Project Properties Window, select Debug tab in LHS and on the right click on “Open debug launch profiles UI” link and change the value of “Hosting Model” in the DropDownList as “Out of Process”.

What is Out of Process Hosting in ASP.NET Core?

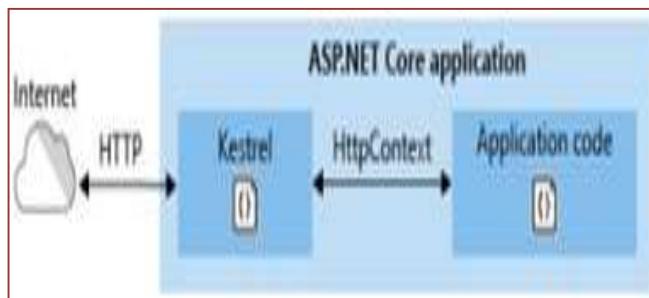
Ans: In case of **ASP.NET Core** => “Out of Process” Hosting Model there will be 2 Web Servers.

1. An internal Web Server which is the Kestrel Web Server.
2. An external Web Server which can be IIS Express or IIS or Apache or Nginx or Tomcat, etc.

The very important point that we need to keep in mind is depending on how you are running your application with the “Out of Process” hosting model, the external **Web Server** may or may not come into picture. As we already discussed that **Kestrel Web Server** is a **Cross-Platform Web Server** that is already embedded with your **ASP.NET Core Application**. So, if we are using “Out of Process” Hosting Model for our **ASP.NET Core Application**, then **Kestrel Web Server** can be used in one of the following ways:

Option 1: we can use **Kestrel Web Server** as an **internet-facing Web Server** which will directly process the incoming **HTTP Requests** and in this scenario only **Kestrel Web Server** is used, and external **Web Server** is not going to be used at all. So, when we run the application using the **.NET Core CLI** then **Kestrel** is the only **Web Server** that is going to be used to handle and process all the incoming **HTTP Requests**. To test this, **open Visual Studio Developer Command Prompt** and run the application as explained earlier.

Now open any browser and navigate to the URL: <http://localhost:5000>, which will display the output on the browser, and in this case even if we specified the **ASP Net Core Hosting Model** as “**OutOfProcess**” also it host the application in **Kestrel** and it only will respond for the incoming requests.



Option 2: The **Kestrel Web Server** can also be used with the combination of a **Reverse Proxy Server** such as **IIS Express**, **IIS**, **Tomcat**, **Apache**, or **Nginx**. When we run our **ASP.NET Core Application** directly from **Visual Studio** choosing **IIS** or **IIS Express Profile**, setting the “**AspNetCoreHostingModel**” element value as “**OutOfProcess**” then **IIS** or **IIS Express** will be used as **Reverse Proxy Server** and **Kestrel** is used as **Internal Web Server**.

To test this go to “**Startup.cs**” file in our project, import the namespace “**Microsoft.AspNetCore.Http**” and then in the “**Configure**” method we find “**app.UseEndpoints**” method call and we need to change the code of it.

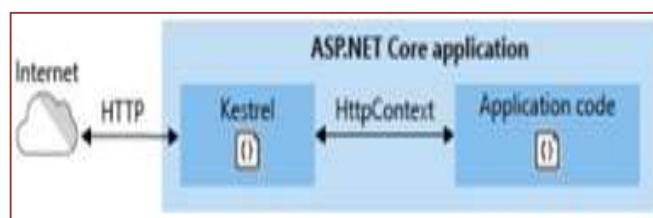
By default, the code in it will be as following:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

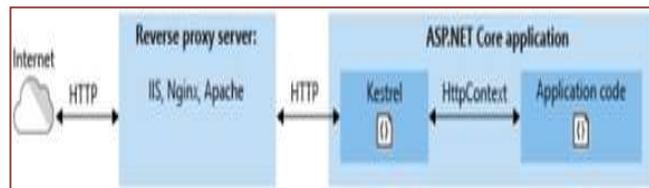
Re-write the code in the method as below:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync(System.Diagnostics.Process.GetCurrentProcess().ProcessName);
    });
});
```

Now set the hosting model as “Out of Process” either in the project property window or in project file and then choose “CoreMVCProject5” profile to run the project which will display the process name as “CoreMVCProject5” only which means the **Kestrel Web Server** is processing our incoming request directly and the URL in the address bar is pointing to **Port No. 5001**, so there is no “Reverse Proxy Server” in usage.



Now choose “IIS” or “IISExpress” profile and run the project which will display the process name as “CoreMVCProject5”, which means the **Kestrel Web Server** is processing our incoming request, and if we watch the URL in the address bar it is pointing to **IIS** or **IIS Express**, so in this case there is a “Reverse Proxy Server” in usage i.e., **IIS Express** or **IIS** which is taking the incoming request and **Kestrel Web Server** which is processing the requests.



Now the question that comes to our mind is, if Kestrel can be used by itself as a Web Server which can directly handle and process the incoming HTTP Request's, then why do we need a reverse proxy server?

Ans: This is all because, the reverse proxy server provides an additional layer of **security** and **configuration** which is not available with **Kestrel Server**, and it also maintains **load balancing**. So, it is a good choice to use **Kestrel Server** along with a **Reverse Proxy Server**. When we use **Kestrel** along with the **Reverse Proxy Server**, then the **Reverse Proxy Server** will receive all the incoming **HTTP Requests** from client's and then forwards that request to the **Kestrel Web Server** for processing. Once the **Kestrel Web Server** process that request, then it sends the response back to the **reverse proxy server** which then sends response back to the requested client over the **internet**.

Working with ASP.NET Core MVC Applications

Create a new “ASP.NET Core Empty” project, naming it as “CoreTestProject”, choose the Target Framework as “.NET 5.0 (Out of Support) or .NET 6.0 (Long Term Support) or .NET 7.0 (Standard Term Support) or .NET 8.0 (Long Term Support)”, select the Checkbox’s “Configure for HTTPS” and “Do not use top-level statements”, and click on “Create” button.

Right now, we have chosen an “Empty Project Template”, so we can configure it as an **ASP.NET Core Web App** or **Web API** or **MVC Application** also. To make it as an **MVC Application**, do the following changes:

ASP.NET Core 5.0 => Open “**Startup.cs**” file and write the below code under “**ConfigureServices**” method of the class:
services.AddControllersWithViews();

ASP.NET Core 6.0 and above => Open “**Program.cs**” file and write the below statement just above the statement “**var app = builder.Build();**”.

builder.Services.AddControllersWithViews();

Now add Static Files and End Points Middleware as below:

ASP.NET Core 5.0 => Now under the **Configure** method of **Startup** class add “**UseStaticFiles**” middleware above “**app.UseRouting**” statement as following:

app.UseStaticFiles();

ASP.NET Core 6.0 and above => Now under **Program** class write the above statement just **below** the statement “**var app = builder.Build();**”.

ASP.NET Core 5.0 => Delete all the code that is present in “**UseEndpoints**” block and write the below code over there which should now look as following:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core 6.0 and above => Now in the bottom of “**Program.cs**” file we find a statement “**app.MapGet("/", () => "Hello World!");**”, comment it or delete it and write the below code over there:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Note: after all the above changes our application is now an “**MVC Application**”, so we can now add “**wwwroot**”, “**Controllers**”, “**Models**” and “**Views**” folders as per our requirements. So let us add all the **4 folders** under our **Project** from **Solution Explorer**.

Adding a Controller in Controllers Folder: Now let’s add a new **Controller** choosing the option “**MVC Controller – Empty**” under “**Add New Scaffolded Item**” window, naming it as “**HomeController**”. If we observe the parent class of “**HomeController**”, it is “**Controller**” only just like in our “**ASP.NET MVC**” but, this class is defined in “**Microsoft.AspNetCore.Mvc**” namespace, whereas in “**ASP.NET MVC**” the class “**Controller**” is defined in “**System.Web.Mvc**” namespace.

In the “**HomeController**” by default we find a “**Action Method**” with the name “**Index**” and the return type of the method is “**IActionResult**” Interface which is the parent of all the “**Action Results**” in an **ASP.NET Core MVC Application**, whereas in **ASP.NET MVC Applications** the parent of all **Action Results** is a class i.e., “**ActionResult**”.

Adding a View to Index Action Method: Now right click on the “**Index**” action method which is present in “**HomeController**” class and select the option “**Add View**” and in the window opened, choose “**Razor View – Empty**” and click on the “**Add**” button which will add a new **View** with the name “**Index.cshtml**” by creating a folder under the “**Views**” folder with the name “**Home**” and the **View** gets added into the “**Home**” folder. Write the below code in it and execute:

```
<h3>Home Controller - Index View</h3>
```

Now hit “**F5**” and run the project which will display a **Message Box** asking about **SSL Certificate** because we have chosen “**Configure for HTTPS**” option while creating the project click on “**Yes**” button which will display a **Security Warning** window asking for installing the certification on your machine click “**Yes**” button again which will launch the **browser** and displays the output.

Note: now if you observe the **URL** in **Browser's - Address Bar**, the protocol will be “**HTTPS**”. In **ASP.NET Core 6.0 and above** the default **Web Server - Visual Studio** uses to run the **Web App's** is **Kestrel** and the **Port** used for **HTTPS** in **Kestrel** will be different and for **HTTP** will be different. We can verify the ports used in “**launchSettings.json**” file. You can even change the **Server** by choosing a **Profile** in **Debug Target Dropdown List** that is present in **Standard Toolbar**.

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 and above**.

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 **Core 5.0** and **Program** class for **Core 6.0 and above**.

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 and above**.

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

To test “Attribute Routing” go to “Startup.cs” file (ASP.NET Core MVC 5.0) of our current project i.e., “CoreTestProject”, delete the code which we have implemented in “UseEndpoints” block earlier and then call “MapControllers” method over there and it should now look as below:

Old Code (Conventional):

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

New Code (Attribute):

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

If you are working with ASP.NET Core MVC 6.0 and above go to Program class, delete or comment the method call “MapControllerRoute” and call the method “MapControllers”.

Old Code (Conventional):

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

New Code (Attribute):

```
app.MapControllers();
```

Note: after doing the above action if we try to run the “Index.cshtml” view we get a “404 – Page Not Found” error because there is no **route pattern** defined anywhere.

To define the routing pattern, go to “Home Controller” class and write the following statements above the “Index” action method which should now look as following:

```
[Route("")]
[Route("Home")]
[Route("Home/Index")]
public IActionResult Index()
{
    return View();
}
```

Note: The above statements when added on the top of action method will be launching the view when we use the corresponding URL's.

We can even set attribute routes on **Controllers** also and to test this write the below statement above the **Controller** class which should now like as following:

```
[Route("Home")]
public class HomeController : Controller
```

In this case we don't require using **controller** name while setting the route for **Index** action method and it should be as following:

```
[Route("/")]
[Route("")]
[Route("Index")]
public IActionResult Index()
```

In **attribute routing** we can hide the original name of the **controller** and **action** method so that end users can't view the name of the **controller** and **action** method and to do that we need to specify an alias name in the route as following:

Specifying an alias name to Controller:

```
[Route("Test")]
public class HomeController : Controller
```

Specifying an alias name to Action Method:

```
[Route("/")]
[Route("")]
[Route("Demo")]
public IActionResult Index()
```

Without hard coding **controller names** or **action method names** in **attribute routes** we can use the concept of "**Token Replacements**", for example we can define the route for **Controller** as:

```
[Route("[controller]")]
public class HomeController : Controller
```

In the above case "**[controller]**" is a token which is replaced with the name of the **Controller** in runtime and same as this we can also do this for action methods also as following:

```
[Route("/")]
[Route("")]
[Route("[action]")]
public IActionResult Index()
```

We can also use "[controller]/[action]" token on Controller class as following:

```
[Route("[controller]/[action]")]
public class HomeController : Controller
```

In this case route for Index action method will be as following:

```
[Route("/")]
[Route("/Home")]
[Route("")]
public IActionResult Index()
```

Route Constraints: we use these to **restrict** the browser requests to match a particular **route** and we can even use **regular expressions** to specify a route constraint.

alpha	=> Accepts only alphabets (Upper or Lower Case)	{x:alpha}
bool	=> Accepts only boolean values	{x:bool}

datetime	=> Accepts only DateTime values	{x:datetime}
decimal	=> Accepts only decimal values	{x:decimal}
double	=> Accepts a 64-bit floating-point value	{x:double}
float	=> Accepts a 32-bit floating-point value	{x:float}
int	=> Accepts a 32-bit integer value	{x:int}
long	=> Accepts a 64-bit integer value	{x:long}
max	=> Accepts a numeric up to a given maximum value	{x:max(10)}
min	=> Accepts a numeric up to a given minimum value	{x:min(5)}
range	=> Accepts a numeric within given range of values	{x:range(10,50)}
length	=> Accepts a string with a specified length	{x:length(6)}
	Or	
maxlength	=> Accepts a specified range of characters	{x:length(1,9)}
minlength	=> Accepts a string with a given maximum length	{x:maxlength(9)}
regex	=> Accepts a string with a given minimum length	{x:minlength(5)}
	=> Matches a given value with a specified regular expression.	{x:regex(^\\d{4}-\\d{4}-\\d{4}\$)}

To practice route constraints first set the route for controller as below:

```
Route("[controller]")
public class HomeController : Controller
```

Define a new action method in the Controller class as following:

```
[Route("Display1/{id?}")]
public string Display1(int id)
{
    return "Value of id is: " + id;
}
```

The above action method gets executed when we use the below URL's:

https://localhost:port/Home/Display1	//Valid
https://localhost:port/Home/Display1/10	//Valid
https://localhost:port/Home/Display1/A	//Valid
https://localhost:port/Home/Display1/false	//Valid
Hello">https://localhost:port/Home/Display1>Hello	//Valid
https://localhost:port/Home/Display1/34.56	//Valid

In the above case we defined "{id?}" as a **route parameter** and it is optional so if we do not pass a value to **id** or if we pass a value of different data types, it will not raise any error but **id** value will be "**0**", whereas if we want the parameter value to be an integer, or any other particular data type or if we want to apply any other restrictions on the action method parameters, we should use **route constraints**.

Add a new action method into the Home Controller class as following:

```
[Route("Display2/{id:int}")]
public string Display2(int id)
{
    return "Value of id is: " + id;
}
```

In the above case we need to call the `Display2` action method by explicitly passing an `integer` value only, whereas if we try to pass any other type of values or if we do not pass a value, we get error.

https://localhost:port/Home/Display2/10	//Valid
https://localhost:port/Home/Display2	//Error
Hello">https://localhost:port/Home/Display2>Hello	//Error
https://localhost:port/Home/Display2/34.56	//Error
https://localhost:port/Home/Display2/false	//Error
https://localhost:port/Home/Display2/A	//Error

Note: if you want to call the above method without passing a value make `int` as `nullable int` i.e., “`int?`”.

```
[Route("Display3/{id:double?}")]
public string Display3(double id)
{
    return "Value of id is: " + id;
}
```

In the above method `id` parameter is defined to accept a `double` value so we can pass an `integer` or a `double` also as following:

https://localhost:port/Home/Display3	//Valid
https://localhost:port/Home/Display3/10	//Valid
https://localhost:port/Home/Display3/34.56	//Valid
https://localhost:port/Home/Display3/A	//Error
Hello">https://localhost:port/Home/Display3>Hello	//Error
https://localhost:port/Home/Display3/false	//Error

```
[Route("Display4/{id:min(50)}")]
public string Display4(int id)
{
    return "Value of id is: " + id;
}
```

The above method is defined with a `constraint` to accept an `integer` with a `minimum value` of `50`, so any value less than `50` throws an `error`:

https://localhost:port/Home/Display4	//Error
https://localhost:port/Home/Display4/-10	//Error
https://localhost:port/Home/Display4/49	//Error
https://localhost:port/Home/Display4/50	//Valid
https://localhost:port/Home/Display4/51	//Valid

```
[Route("Display5/{id:max(100)}")]
public string Display5(int id)
{
    return "Value of id is: " + id;
}
```

The above **method** is defined with a **constraint** to accept an **integer** with a **maximum value of 100**, so any value greater than **100** throws an **error**:

https://localhost:port/Home/Display5	//Error
https://localhost:port/Home/Display5/99	//Valid
https://localhost:port/Home/Display5/100	//Valid
https://localhost:port/Home/Display5/101	//Error
https://localhost:port/Home/Display5/-10	//Valid

```
[Route("Display6/{id:range(51, 100)}")]
public string Display6(int id)
{
    return "Value of id is: " + id;
}
```

The above **method** is defined with a **constraint** to accept an **integer** within a **range of 51 to 100**, so any value less than **51** and greater than **100** throws an **error**:

http://localhost:port/Home/Display6	//Error
http://localhost:port/Home/Display6/50	//Error
http://localhost:port/Home/Display6/51	//Valid
http://localhost:port/Home/Display6/100	//Valid
http://localhost:port/Home/Display6/101	//Error

```
[Route("Display7/{name:length(5)}")]
public string Display7(string name)
{
    return "Name of the user is: " + name;
}
```

The above **method** is defined with a **constraint** to accept a **string** value with a **length of 5 characters**, so any value less than **5** or greater than **5** characters, throws an **error**:

http://localhost:port/Home/Display7	//Error
http://localhost:port/Home/Display7/Abcd	//Error
http://localhost:port/Home/Display7/Abcde	//Valid
http://localhost:port/Home/Display7/abcdef	//Error
http://localhost:port/Home/Display7/false	//Valid
http://localhost:port/Home/Display7/12345	//Valid

```
[Route("Display8/{name:length(3, 10)}")]
public string Display8(string name)
{
    return "Name of the user is: " + name;
}
```

The above **method** is defined with a **constraint** to accept a **string** value with a **minimum length of 3** characters and **maximum length of 10** characters, so any value less than **3** characters or greater than **10** characters, throws an **error**:

http://localhost:port/Home/Display8	//Error
http://localhost:port/Home/Display8/AB	//Error
http://localhost:port/Home/Display8/Sai	//Valid
http://localhost:port/Home/Display8/Venkat	//Valid
http://localhost:port/Home/Display8/Bangarraju	//Valid
http://localhost:port/Home/Display8/DavidWarner	//Error

```
[Route("Display9/{name:minlength(3)}")]
public string Display9(string name)
{
    return "Name of the user is: " + name;
}
```

The above **method** is defined with a **constraint** to accept a **string** value with a **minimum** length of **3** characters, so any value less than **3** characters, throws an **error**:

http://localhost:port/Home/Display9	//Error
http://localhost:port/Home/Display9/AB	//Error
http://localhost:port/Home/Display9/Sai	//Valid
http://localhost:port/Home/Display9/Bangarraju	//Valid
http://localhost:port/Home/Display9/DavidWarner	//Valid

```
[Route("Display10/{name:maxlength(10)}")]
public string Display10(string name)
{
    return "Name of the user is: " + name;
}
```

The above **method** is defined with a **constraint** to accept a **string** value with a **maximum** length of **10** characters, so any value greater than **10** characters, throws an **error**:

http://localhost:port/Home/Display10	//Error
http://localhost:port/Home/Display10/AB	//Valid
http://localhost:port/Home/Display10/Sai	//Valid
http://localhost:port/Home/Display10/Bangarraju	//Valid
http://localhost:port/Home/Display10/DavidWarner	//Error

```
[Route("Display11/{name:alpha}")]
public string Display11(string name)
{
    return "Name of the user is: " + name;
}
```

The above **method** is defined with a **constraint** to accept only **string** values so any value other than **string**, throws an **error**:

http://localhost:port/Home/Display11	//Error
http://localhost:port/Home/Display11/Raju	//Valid
http://localhost:port/Home/Display11/1234	//Error

http://localhost:port/Home/Display11/34.56	//Error
http://localhost:port/Home/Display11/false	//Valid

```
[Route("Display12/{flag:bool}")]
public string Display12(bool flag)
{
    if (flag)
        return "Hello India!";
    else
        return "Hello World!";
}
```

The above **method** is defined with a **constraint** to accept only **boolean** values so any value other than **boolean**, throws an **error**:

http://localhost:port/Home/Display12	//Error
http://localhost:port/Home/Display12/true	//Valid
Hello">http://localhost:port/Home/Display12>Hello	//Error
http://localhost:port/Home/Display12/34.56	//Error
http://localhost:port/Home/Display12/false	//Valid

```
[Route("Display13/{aadhar:regex(^\\d{4}-\\d{4}-\\d{4}$)")]
public string Display13(string aadhar)
{
    return "Aadhar Id of the user is: " + aadhar;
}
```

The above **method** is defined with a **constraint** to accept Aadhar Id by using a **regular expression**, so any value that doesn't match with the **expression** will throw an **error**:

http://localhost:port/Home/Display13	//Error
http://localhost:port/Home/Display13/1234-1234-1234	//Valid
http://localhost:port/Home/Display13/1234-1234-123	//Error
http://localhost:port/Home/Display13/1234-1234-12345	//Error
http://localhost:port/Home/Display13/1234-1234=1234	//Error

```
[Route("Display14/{id}/{name}")]
public string Display14(int id, string name)
{
    return "Id of the user is: " + id + " and name of the user is: " + name;
}
```

The above **method** is defined with 2 **mandatory** parameters **id** and **name**, so if we try to call it with any **other** value other than those it will throw an **error**:

http://localhost:50293/Home/Display14	//Error
http://localhost:50293/Home/Display14/101	//Error
http://localhost:50293/Home/Display14/101/Raju	//Valid
http://localhost:50293/Home/Display14/101/Raju/50.56	//Error

Creating Views (User Interfaces) in MVC CORE

Create a new “ASP.NET Core Empty” project naming it as “MVCCoreTagHelpers”, choose Target Framework as “.NET 5.0 (Out of Support)” or “.NET 6.0 (Long Term Support)” or “.NET 7.0 (Standard Term Support)” or “.NET 8.0 (Long Term Support)” and click on the Create button. Because we have chosen an empty project template, configure the project to work as an **MVC Application** as explained earlier and then do the following:

Add a new folder under the project naming it as “**Models**”, add a new class in to that folder naming it as “**LoginModel**” and write the below code in it importing “**System.ComponentModel.DataAnnotations**” namespace:

```
public class LoginModel
{
    [EmailAddress]
    public string Email { get; set; }
    [DataType(DataType.Password)]
    public string Password { get; set; }
    public bool RememberMe { get; set; }
}
```

Add a new folder under the project naming it as “**Controllers**”, add a new Controller class into the folder naming it as “**AccountController**”, and write the below code in it by deleting the existing **Index** action method:

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

Add a view to “**Login**” action method and to do that right click on the method that is present in the Controller class, select the option “**Add View**” which launches a window, in that select “**Razor View-Empty**”, click on “**Add**” button which opens a window asking for a name, enter name as “**Login.cshtml**” and click “**Add**” button.

Note: now in the view we need to import the **Model** class to access the properties of **Model** class, as following:

```
@model MVCCoreTagHelpers.Models.LoginModel
```

In the above statement we are prefixing the “**Namespace-Name**” before the “**Model Class-Name**”, and we need to do this in every **View** where we are using “**Model Binding**”. To simply the coding or decrease the typing work **MVC Core** introduced an option called as “**ViewImports**” i.e., it is a new “**.cshtml**” file present under the **Views** folder just like “**_ViewStart.cshtml**”, but with the name “**_ViewImports.cshtml**” and we can use this file for **importing** all the required **namespaces** for **Views**, so that we don’t require to do that again and again in each **View**.

If we create a project of type “**ASP.NET Core Empty**” then we will not have “**_ViewImports.cshtml**” by default in our project so we need to explicitly add it, and to do that right click on the **Views** folder and select the option **Add => View** which will open a new window and in that window select “**Razor View Empty**” and click on the “**Add**” button which opens another window and in that window select the template “**Razor View Imports**” which displays “**_ViewImports.cshtml**” as the name of file, click on “**Add**” button to add it to the **Project**. By default, the file will be empty and, in the file, write the below statement:

```
@using MVCCoreTagHelpers.Models
```

Note: If we create a project of type “ASP.NET Core Web App (Model-View-Controller)”, then we don’t require doing all the above things because my default “_ViewImports.cshtml” file is added, and the **Models** namespace is also imported.

Now in “Login.cshtml” without prefixing the namespace name before **Model** class name we can directly write it as following:

```
@model LoginModel
```

Installing Bootstrap for designing Views: if we want to use bootstrap for styling of elements, we need to first install bootstrap in our projects and we can do that either by using “NuGet Package Manager” or “Library Manager” for managing all the **client-side** libraries.

Working with Library Manager: Right-click on the **Project** in **Solution Explorer** and choose **Add => Client-Side Library**, which launches “Add Client-Side Library” dialog over there, and in that window fill the following details:

- **Provider:** cdnjs (default)
- **Library:** [twitter-bootstrap@5.3.3](#) (5.3.3 is the latest version by the time of preparing this document).
- Select Include all library files radio button.
- **Target Location:** wwwroot/lib/bootstrap

Click install and this will install **bootstrap** in our project under the specified target location and this will also add a file in the project with the name “libman.json” and under this file the details of all the **libraries** installed will be present.

Now write the below code in the “Login.cshtml” file by deleting all the content in the file except the **Model** class import statement, for creating the **Login Form** using “**HtmlHelpers**”:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Login Form</title>
    <link href="~/lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
  </head>
  <body>
    <h1>Login Form</h1>
    <section>
      @using (Html.BeginForm("Login", "Account", FormMethod.Post, new { @class = "form-horizontal", role = "form" }))
      {
        @Html.AntiForgeryToken()
        <hr />
        <div class="form-group">
          @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
          @Html.TextBoxFor(m => m.Email, new { @class = "col-md-4 form-control" })
        </div>
        <div class="form-group">
          @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
          @Html.PasswordFor(m => m.Password, new { @class = "col-md-4 form-control" })
        </div>
      }
    </section>
  </body>
</html>
```

```

</div>
<div class="form-group">
    @Html.CheckBoxFor(m => m.RememberMe)
    @Html.LabelFor(m => m.RememberMe)
</div>
<div class="form-group">
    <input type="submit" value="Log In" class="btn btn-primary" />
    <input type="reset" value="Reset" class="btn btn-primary" />
</div>
}
</section>
</body>
</html>

```

In the View what we have created above, we used “**Html Helpers**” to design the UI or **View** whereas “**MVC Core**” provided us “**Tag Helpers**” for UI or **View** designing.

How to use Tag Helpers?

Ans: To use the **In-built Tag Helpers**, we must import them by using “`@addTagHelper`” directive in “`_ViewImports.cshtml`” file. If our project is “**ASP.NET Core Empty**” we need to perform importing explicitly whereas if the project is “**ASP.NET Core Web App (Model-View-Controller)**” then it is automatically imported and to check that go to “`_ViewImports.cshtml`” and watch the code, and there we find the below statement and if not found do it manually:

```
@addTagHelper *,Microsoft.AspNetCore.Mvc.TagHelpers
```

Note: The above statement will import all the **Tag helpers** that are present in the namespace “`Microsoft.AspNetCore.Mvc.TagHelpers`” and as discussed above if we import a library in “`_ViewImports.cshtml`” file then it will be available for all the views. If you don’t want any views to use globally imported **Tag Helpers**, you can use `@removeTagHelper` in the respective view.

If we want to use these **Tag Helpers** in our **Login Page**, then delete all the content that is present inside of `<section></section>` tags in the code present there and write the below code inside of `<section></section>` tags:

```

<form asp-controller="Account" asp-action="Login">
    <hr />
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="col-md-4 form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="col-md-4 form-control" />
    </div>
    <div class="form-group">
        <input asp-for="RememberMe" />
        <label asp-for="RememberMe"></label>
    </div>

```

```

<div class="form-group">
    <input type="submit" value="Log In" class="btn btn-primary" />
    <input type="reset" value="Reset" class="btn btn-primary" />
</div>
</form>

```

MVC Core Tag Helpers: these will enable **server-side code** to participate in creating and rendering **HTML** elements in **Razor Pages**. There are many built-in **Tag Helpers** for common tasks - such as creating **forms**, **links**, **loading assets** and more - and even more available in public **GitHub** repositories as **NuGet** packages as well as we can also create our own **Tag Helpers**.

Tag Helpers are authored in **C#**, and they target **HTML Elements** based on the element or attribute name. For example, the built-in “**LabelTagHelper**” class targets the **HTML Label Element** when the “**LabelTagHelper**” attributes are applied. If we are familiar with **HTML Helpers** in **MVC 5**, **Tag Helpers** will reduce the explicit transitions between **HTML** and **C#** in **Razor views** i.e., in many cases, **HTML Helpers** provide an alternative approach to a specific **Tag Helper**, but it’s important to recognize that **Tag Helpers** don’t replace **HTML Helpers** and there is no **Tag Helper** available for every **HTML Helper**.

What Tag Helpers provide?

Ans: Tag Helpers provide an **HTML friendly development experience** for the most part, **Razor** markup using **Tag Helpers** looks like standard **HTML**. Front-end designers who are familiar with **HTML/CSS/JavaScript** can edit **Razor** without learning **C# Razor** syntax.

Tag Helpers are a way to make developers more productive and be able to produce more **robust**, **reliable**, and **maintainable** code using information that is available on the server. For example, historically the mantra of updating images was to change the name of image whenever we change the image because Images will be **cached** by the browsers for performance reasons, and unless we change the name of an image, you risk clients getting a **stale copy**. Historically, after an image was edited, the name must be changed and each reference to the image in the web application needed to be updated. Not only this is labor intensive, but it’s also errors prone (you could miss a reference, accidentally enter the wrong string, etc.). The built-in “**ImageTagHelper**” can do this work automatically for us i.e., this will append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image, so clients are guaranteed to get the latest image. This robustness and labor savings comes essentially free by using the “**ImageTagHelper**”.

All the built-in **Tag Helpers** target the standard **HTML Elements** and provide server-side attributes for those elements. For example, the **<input>** element used in our views contains the “**asp-for**” attribute and this attribute extracts the name of the specified **Model Property** into the rendered **HTML**.

What are Tag Helpers?

Ans: Tag Helpers are **Classes** written in **C#** but are attached to **HTML Elements** to run server-side code from **Razor Views**, in other words, **Views** that are created in **HTML** do have their presentation logic defined in **C#**, which is ultimately executed on **Web Server**. Examples of some built-in Tag Helpers are **Anchor tag**, **Environment tag**, etc.

Form TagHelper: this is used for creating Form element.

Tag helper we use:

```
<form asp-controller="Home" asp-action="Index">
    <!-- Input and Submit elements -->
</form>
```

Rendered HTML will be as following:

```
<form method="post" action="/Home/Index">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<generated value>">
</form>
```

Note: MVC runtime generates action attribute value based on the **Form Tag Helper Attributes** => **asp-controller** and **asp-action**. The **Form Tag Helper** also generates a hidden **Request Verification Token** to prevent cross-site request forgery (when used with **[ValidateAntiForgeryToken]** filter attribute in **HTTP Post** action method). Protecting a pure **HTML Form** from **cross-site request forgery** is difficult; the **Form Tag Helper** provides this service **free** for you. The default method is **“Post”** even if no specified.

Using a named route: The **asp-route** Tag Helper attribute can also generate markup for the HTML action attribute. An application with a route named **“RegisterRoute”** could use the following markup for the Registration Page:

```
public class AccountController : Controller
{
    [Route("/Account/Register", Name = "RegisterRoute")]
    public ActionResult Registration()
    {
        return "This is a Registration Page.";
    }
}
```

Tag helper we use:

```
<form asp-route="RegisterRoute">
    <!-- Input and Submit elements -->
</form>
```

This renders the below HTML:

```
<form method="post" action="/Account/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<generated value>">
</form>
```

Form Action Tag Helper: This generates the form action attribute based on the **Button** or **Input** type - **image** tags. The form action attributes controls where a form submits its data. It binds to **<input>** elements of type **image** and **<button>** elements. The Form Action Tag Helper enables the usage of several **“AnchorTagHelper” “asp- attributes”** to control what **“form action”** link is generated for the corresponding element. For example, if we define a Controller class with a set of action methods using attribute routing along with a name to the route then we can use form action tag helper to generate submit links.

```

public class AccountController : Controller
{
    [Route("/Account/Login", Name = "LoginRoute")]
    public string Login()
    {
        return "This is a Login Page.";
    }
    [Route("/Account/Register", Name = "RegisterRoute")]
    public string Register()
    {
        return "This is a Registration Page.";
    }
}

```

Submit to controller's action method example:

```

<form method="post">
    //Place controls for Login
    <button asp-controller="Account" asp-action="Login">Login</button>
    //Place controls for Register
    <button asp-controller="Account" asp-action="Register">Register</button>
</form>

```

Submit to route example:

```

<form method="post">
    //Place controls for Login
    <button asp-route="LoginRoute">Login</button>
    //Place controls for Register
    <button asp-route="RegisterRoute">Register</button>
</form>

```

The Input Tag Helper: This renders an Html input element.

```
<input asp-for="<Expression Name>">
```

Note: The **Input Tag Helper** sets the HTML “**type**” attribute based on the **.NET type**. The below table lists, some common .NET types, and generated HTML types:

.NET Type	Input Type
bool	type="checkbox"
string	type="text"
DateTime	type="datetime-local"
Byte, int, single, double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types:

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"

[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

The Select Tag Helper: Generates select and associated option elements for properties of your model. The Select Tag Helper's "asp-for" specifies the model property name for the select element and "asp-items" specifies the option for elements. For example:

```
public class CountryModel
{
    public string Country { get; set; }
    public List<SelectListItem> Countries { get; } = new List<SelectListItem>
    {
        new SelectListItem { Value = "C1", Text = "Delhi" },
        new SelectListItem { Value = "C2", Text = "Kolkata" },
        new SelectListItem { Value = "C3", Text = "Mumbai" },
        new SelectListItem { Value = "C4", Text = "Chennai" },
        new SelectListItem { Value = "C5", Text = "Bengaluru" },
        new SelectListItem { Value = "C6", Text = "Hyderabad" }
    };
}
```

Tag helper we use: <select asp-for="Country" asp-items="Model.Countries"></select >

Multi-select: The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the "asp-for" attribute is an "IEnumerable". For example, given the following model:

```
public class CountryModel
{
    public IEnumerable<string> Country { get; set; }
    public List<SelectListItem> Countries { get; } = new List<SelectListItem>
    {
        new SelectListItem { Value = "C1", Text = "Delhi" },
        new SelectListItem { Value = "C2", Text = "Kolkata" },
        new SelectListItem { Value = "C3", Text = "Mumbai" },
        new SelectListItem { Value = "C4", Text = "Chennai" },
        new SelectListItem { Value = "C5", Text = "Bengaluru" },
        new SelectListItem { Value = "C6", Text = "Hyderabad" }
    };
}
```

Tag helper we use: <select asp-for="Country" asp-items="Model.Countries"></select >

Anchor Tag Helper: Anchor Tag Helper enhances the standard HTML anchor (<a ... >) tag by adding new attributes.

```

public class StudentController : Controller
{
    [Route("/Student/GetStudent", Name = "StudentGet")]
    public ActionResult GetStudent()
    {
        return View();
    }
}

```

Tag helper we use: `<a asp-controller="Student" asp-action="GetStudent">Display all Students`

asp-route: The `asp-route` attribute is used for creating a URL link directly to a named route. Using routing attributes, a route can be named as shown in the “`StudentController`” and used in its “`GetStudent`” action:

Tag helper we use: `<a asp-route="StudentGet">Student Details`

Image Tag Helper: The Image Tag Helper enhances the “``” tag to provide **cache-busting** behavior for static image files. Cache-busting string means a unique value representing the hash of the static image file appended to the asset’s **URL**. The unique string prompts clients (and some proxies) to reload the image from the host web server and not from the client’s cache.

Tag helper we use: ``

Environment Tag Helper: The Environment Tag Helper conditionally renders its enclosed content based on the current **hosting environment** i.e., this **Tag Helper’s include or exclude attributes** can be used to specify a comma-separated list of environment names. If any of the given environment names match the current environment, the enclosed content is rendered.

To work with “**Image**” and “**Environment**” TagHelpers add 2 new action methods in “`AccountController`” class of “`MVCCoreTagHelpers`” project and write the below code under the existing Login method:

```

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

public ViewResult Register()
{
    return View();
}

```

To work with “**ImageTagHelper**” add a new folder with the name “**images**” under “**wwwroot**” folder and copy an image into it with the name “**Register.jpg**”. Add a **View** to **Register** action method and write the below code in it by deleting the exiting content in it:

```

<!DOCTYPE html>
<html>
    <head>

```

```

<title>Registration Form</title>
<link href="~/lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
    <h1>Registration Form</h1>
    
</body>
</html>

```

Call the Register action method to launch “Register.cshtml” file, now replace the image in images folder with a new image but the image name should be “Register.jpg” only and launch Register view again but you will see old image only but not the new image because of the **cache busting** behavior of the browser and to resolve the problem add “asp-append-version=”true”” to the image tag which should now look as following:

```

```

When we add the new attribute to Image Tag it will generate a unique hash value based on the image content and appends to the image name, so whenever image changes (not the name) a new hash value will be generated, appended to the image name, and then loads the image from the server without loading it from browser cache.

To work with **EnvironmentTagHelper** add a new view to **Index** action method and write the below code in it by deleting the exiting content in it:

```

@inject Microsoft.AspNetCore.Hosting.IWebHostEnvironment env
<!DOCTYPE html>
<html>
    <head>
        <title>Index Form</title>
        <environment include="Development">
            <link href="~/lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
        </environment>
        <environment exclude="Development">
            <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.1.1/css/bootstrap.min.css"
                  crossorigin="anonymous" referrerpolicy="no-referrer"
                  integrity="sha512-6KY5s6UI5J7SVYuZB4S/CZMyPylqyyNZco376NM2Z8Sb8OxEdp02e1jkKk/wZxIEmjQ6DRCEBhni+gpr9c4tvA==" />
        </environment>
    </head>
    <body>
        <h3>Environment: @env.EnvironmentName</h3>
    </body>
</html>

```

The Environment Tag Helper conditionally renders its enclosed content based on the current hosting environment and to test this, run the Index view and in the browser use “**View Page Source**” option, and watch the “**<link>**” tag in “**<head>**” section which will be loading the assets from “**local folder**” because our project is right now running in “Development” environment and to change it open “**launchSettings.json**” file present under Properties and change (“**ASPNETCORE_ENVIRONMENT**”: “**Development**”) as (“**ASPNETCORE_ENVIRONMENT**”: “**Production**”)

and run Index action method again and watch the “<link>” tag by using “View Page Source” which will now load the assets from “CDN”.

Custom Tag Helpers: We can also develop our own **Tag Helpers** and consume them in our applications and to do that we need to follow the below process:

- Step 1:** Add a new folder under the project with the name “**TagHelpers**”.
 - Step 2:** Define a new class inheriting from the pre-defined class “**TagHelper**” which is present under the namespace “**Microsoft.AspNetCore.Razor.TagHelpers**”. The name of the class should be same as the **tag** we are targeting suffixed with **TagHelper**.
 - Step 3:** Now under the class define all the **properties** we are expecting for our **tag**.
 - Step 4:** Implement logic under the class by overriding the method “**Process**” of “**TagHelper**” class.
 - Step 5:** Import the project namespace in “**_ViewImports.cshtml**” file.
 - Step 6:** Start consuming the new **TagHelper** class in the required Views.
-

Creating a Custom Tag Helper:

Step 1: Add a new folder under the project naming it as “**TagHelpers**” and in to that folder add a new class naming it as “**EmailTagHelper.cs**” and write the below code in it by importing “**Microsoft.AspNetCore.Razor.TagHelpers**” namespace:

```
public class EmailTagHelper : TagHelper
{
    public string MailTo { get; set; }
    public string DomainName { get; set; }
    public string TargetAddress { get; set; }
    public string InnerHtml { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";

        if (!String.IsNullOrEmpty(TargetAddress))
            output.Attributes.SetAttribute("href", $"mailto:{TargetAddress}");
        else
            output.Attributes.SetAttribute("href", $"mailto:{MailTo}@{DomainName}");

        if (!String.IsNullOrEmpty(InnerHtml))
            output.Content.SetContent(InnerHtml);
        else if (!String.IsNullOrEmpty(TargetAddress))
            output.Content.SetContent(TargetAddress);
        else
            output.Content.SetContent($"'{MailTo}@{DomainName}'");
    }
}
```

In the above case we are targeting the “<email>” tag and implementing the logic so whenever we use the “<email>” tag in our **View** file it will be changing to “<a>” tag and the 4 properties we defined in the class (**MailTo**, **DomainName**, **TargetAddress** and **InnerHtml**) will be defining the way how “<a>” tag is rendered.

Step 2: go to “_ViewImports.cshtml” file and import the project namespace on top of the file as following:

```
@addTagHelper *, MVCCoreTagHelpers
```

Step 3: Write the below code in <body> tag of “Index.cshtml” file by deleting the existing code present there:

```
<h3>Click to navigate:</h3>
<a class="btn btn-primary" href="/Home/Login">Login</a>
@Html.ActionLink("Register", "Register", "Home", null, new { @class="btn btn-primary" })
<a class="btn btn-primary" asp-controller="Home" asp-action="About">About</a>
<br /><br />
<strong>Support:</strong>
<email target-address="support@nareshit.com"></email><br />
<strong>Marketing:</strong>
<email mail-to="marketing" domain-name="nareshit.com"></email><br />
<strong>Sales:</strong>
<email mail-to="sales" domain-name="nareshit.com" inner-html="Contact"></email>
```

Note: Now run the “Index.cshtml” file and watch the output of the “<email>” tag by using “View Page Source” option where we notice “<a>” tags getting generated for all “<email>” tags.

Data Management using XML

Create a new “ASP.NET Core Web App (Model-View-Controller)” naming it as “MVCDHProject”, choose Target Framework as “.NET 8.0 (Long Term Support)”, and click on the **Create** button. Add a new **XML File** in the **Project** naming it as “Customer.xml” and write the below code in it:

```
<Customers>
  <Customer>
    <Custid>101</Custid>
    <Name>Scott</Name>
    <Balance>25000</Balance>
    <City>Hyderabad</City>
    <Status>True</Status>
  </Customer>
  <Customer>
    <Custid>102</Custid>
    <Name>Smith</Name>
    <Balance>35000.00</Balance>
    <City>Chennai</City>
    <Status>True</Status>
  </Customer>
  <Customer>
    <Custid>103</Custid>
    <Name>David</Name>
    <Balance>45000.00</Balance>
    <City>Bengaluru</City>
```

```

<Status>True</Status>
</Customer>
</Customers>

```

Add a new class in to the Models folder naming it as “Customer” and write the below code in it:

```

public class Customer
{
    public int Custid { get; set; }
    public string? Name { get; set; }
    public decimal? Balance { get; set; }
    public string? City { get; set; }
    public bool Status { get; set; }
}

```

Add another new class in to the Models folder naming it as “CustomerXmlDAL” and write the below code in it:

```

using System.Data;
public class CustomerXmlDAL
{
    DataSet ds;
    public CustomerXmlDAL()
    {
        ds = new DataSet();
        ds.ReadXml("Customer.xml");
        //Adding Primary Key on Custid of DataTable
        ds.Tables[0].PrimaryKey = new DataColumn[] { ds.Tables[0].Columns["Custid"] };
    }
    public List<Customer> Customers_Select()
    {
        List<Customer> customers = new List<Customer>();
        foreach (DataRow dr in ds.Tables[0].Rows)
        {
            Customer obj = new Customer
            {
                Custid = Convert.ToInt32(dr["Custid"]),
                Name = (string)dr["Name"],
                Balance = Convert.ToDecimal(dr["Balance"]),
                City = (string)dr["City"],
                Status = Convert.ToBoolean(dr["Status"])
            };
            customers.Add(obj);
        }
        return customers;
    }
    public Customer Customer_Select(int Custid)
    {

```

```

//Finding a DataRow based on its Primary Key value
DataRow dr = ds.Tables[0].Rows.Find(Custid);
Customer customer = new Customer
{
    Custid = Convert.ToInt32(dr["Custid"]),
    Name = Convert.ToString(dr["Name"]),
    Balance = Convert.ToDecimal(dr["Balance"]),
    City = Convert.ToString(dr["City"]),
    Status = Convert.ToBoolean(dr["Status"])
};
return customer;
}
public void Customer_Insert(Customer customer)
{
    //Creating a new DataRow based on the DataTable structure
    DataRow dr = ds.Tables[0].NewRow();
    //Assigning values to each Column of the DataRow
    dr["Custid"] = customer.Custid;
    dr["Name"] = customer.Name;
    dr["Balance"] = customer.Balance;
    dr["City"] = customer.City;
    dr["Status"] = customer.Status;
    //Adding the new DataRow to DataTable
    ds.Tables[0].Rows.Add(dr);
    //Saving data back to XML file
    ds.WriteXml("Customer.xml");
}
public void Customer_Update(Customer customer)
{
    //Finding a DataRow based on its Primary Key value
    DataRow dr = ds.Tables[0].Rows.Find(customer.Custid);
    //Finding the Index of DataRow by calling IndexOf method
    int Index = ds.Tables[0].Rows.IndexOf(dr);
    //Overriding the old values in DataRow with new values based on the Index
    ds.Tables[0].Rows[Index]["Name"] = customer.Name;
    ds.Tables[0].Rows[Index]["Balance"] = customer.Balance;
    ds.Tables[0].Rows[Index]["City"] = customer.City;
    //Saving data back to XML file
    ds.WriteXml("Customer.xml");
}
public void Customer_Delete(int Custid)
{
    //Finding a DataRow based on its Primary Key value
    DataRow dr = ds.Tables[0].Rows.Find(Custid);
    //Finding the Index of DataRow by calling IndexOf method
    int Index = ds.Tables[0].Rows.IndexOf(dr);
}

```

```

//Deleting the DataRow from DataTable by using Index
ds.Tables[0].Rows[Index].Delete();
//Saving data back to XML file
ds.WriteXml("Customer.xml");
}
}

```

Add a new Controller into Controllers folder naming it as “CustomerController” and write the below code in it:

```

using MVCDHProject.Models;
public class CustomerController : Controller
{
    CustomerXmlDAL obj = new CustomerXmlDAL();

    public ViewResult DisplayCustomers()
    {
        return View(obj.Customers_Select());
    }
    public ViewResult DisplayCustomer(int Custid)
    {
        return View(obj.Customer_Select(Custid));
    }
    public ViewResult AddCustomer()
    {
        return View();
    }
    [HttpPost]
    public RedirectToActionResult AddCustomer(Customer customer)
    {
        obj.Customer_Insert(customer);
        return RedirectToAction("DisplayCustomers");
    }
    public ViewResult EditCustomer(int Custid)
    {
        return View(obj.Customer_Select(Custid));
    }
    public RedirectToActionResult UpdateCustomer(Customer customer)
    {
        obj.Customer_Update(customer);
        return RedirectToAction("DisplayCustomers");
    }
    public RedirectToActionResult DeleteCustomer(int Custid)
    {
        obj.Customer_Delete(Custid);
        return RedirectToAction("DisplayCustomers");
    }
}

```

Add a View to DisplayCustomers Action method and write the below code by deleting existing code over there:

```
@model IEnumerable<Customer>
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" align="center" class="table-condensed">
<tr>
<th>@Html.DisplayNameFor(C => C.Custid)</th>
<th>@Html.DisplayNameFor(C => C.Name)</th>
<th>@Html.DisplayNameFor(C => C.Balance)</th>
<th>@Html.DisplayNameFor(C => C.City)</th>
<th>@Html.DisplayNameFor(C => C.Status)</th>
<th>Actions</th>
</tr>
@foreach (Customer customer in Model)
{
<tr>
<td align="center">@Html.DisplayFor(C => customer.Custid)</td>
<td>@Html.DisplayFor(C => customer.Name)</td>
<td>@Html.DisplayFor(C => customer.Balance)</td>
<td>@Html.DisplayFor(C => customer.City)</td>
<td align="center">@Html.DisplayFor(C => customer.Status)</td>
<td>
<a asp-action="DisplayCustomer" asp-route-Custid="@customer.Custid">View</a> &nbsp;
<a asp-action="EditCustomer" asp-route-Custid="@customer.Custid">Edit</a> &nbsp;
<a asp-action="DeleteCustomer" asp-route-Custid="@customer.Custid"
onclick="return confirm('Are you sure of deleting the record?')">Delete</a>
</td>
</tr>
}
<tr><td colspan="6" align="center"><a asp-action="AddCustomer">Add New Customer</a></td></tr>
</table>
```

Add a View to DisplayCustomer Action method and write the below code by deleting existing code over there:

```
@model Customer
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" align="center">
<tr><td>Custid:</td><td>@Model.Custid</td></tr>
<tr><td>Name:</td><td>@Model.Name</td></tr>
<tr><td>Balance:</td><td>@Model.Balance</td></tr>
<tr><td>City:</td><td>@Model.City</td></tr>
<tr><td>Status:</td><td>@Model.Status</td></tr>
</table>
<div style="text-align:center">
<a asp-action="DisplayCustomers" align="center">Back to Customer Details</a>
</div>
```

Add a View to AddCustomer Action method and write the below code by deleting existing code over there:

```
@model Customer
<form asp-controller="Customer" asp-action="AddCustomer">
    <div><label asp-for="Custid"></label><br /><input asp-for="Custid" /></div>
    <div><label asp-for="Name"></label><br /><input asp-for="Name" /></div>
    <div><label asp-for="Balance"></label><br /><input asp-for="Balance" /></div>
    <div><label asp-for="City"></label><br /><input asp-for="City" /></div>
    <div><label asp-for="Status"></label><br /><input asp-for="Status" /></div>
    <div><input type="submit" value="Save" /><input type="reset" value="Reset" /></div>
    <div><a asp-action="DisplayCustomers" align="center">Back to Customer Details</a></div>
</form>
```

Add a View to EditCustomer Action method and write the below code by deleting existing code over there:

```
@model Customer
<form asp-controller="Customer" asp-action="UpdateCustomer">
    <div><label asp-for="Custid"></label><br /><input asp-for="Custid" readonly /></div>
    <div><label asp-for="Name"></label><br /><input asp-for="Name" /></div>
    <div><label asp-for="Balance"></label><br /><input asp-for="Balance" /></div>
    <div><label asp-for="City"></label><br /><input asp-for="City" /></div>
    <div><label asp-for="Status"></label><br /><input asp-for="Status" disabled /></div>
    <div>
        <input type="submit" value="Update" />
        <a asp-action="DisplayCustomers" align="center">Cancel</a>
    </div>
</form>
```

What is the Problem in the above implementation?

Ans: As we can see in the above “CustomerController” class, to get the **Customer** Data, **Controller** depends on “CustomerXmlDAL” class. So, with-in the **Controller** class we have created the instance of “CustomerXmlDAL” class and then invoking all the methods of that class as per our requirements. So, there is a **tight coupling** between “CustomerController” class and “CustomerXmlDAL” classes. Tomorrow if the **DAL** implementation class is changed then we also need to change the code in all the **Controller** classes where we used “CustomerXmlDAL” because **Controller** and **DAL** are **tightly coupled** with each other.

Note: We can overcome this problem with the help of “Dependency Injection Design Pattern”.

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** or **Implementation Classes** that provides services to the **Client**.
- **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 service.

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 from ASP.NET Core 6.0 and above. 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>() | Or | AddSingleton(type Interface, type Class) |
| ➤ AddScoped<Interface, Class>() | Or | AddScoped(type Interface, type Class) |
| ➤ AddTransient<Interface, Class>() | Or | AddTransient(type Interface, type Class) |

Note: The built-in "IoC" container manages the lifetime of a registered service i.e., it automatically disposes the 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 (one 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 in a request).

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.

Note: we can also manually inject dependency object i.e., service into client class by implementing the below code:

```
var services = HttpContext.RequestServices;
var obj = (<Interface_Name>)services.GetService(typeof(<Interface_Name>));
```

To implement “Dependency Injection” into our application i.e., “MVCDHProject” do the following:

Step 1: Define an interface under Model’s folder with the name “ICustomerDAL” and write the below code in it.

```
public interface ICustomerDAL
{
    List<Customer> Customers_Select();
    Customer Customer_Select(int Custid);
    void Customer_Insert(Customer customer);
    void Customer_Update(Customer customer);
    void Customer_Delete(int Custid);
}
```

Step 2: Go to “CustomerXmlDAL” class and make “ICustomerDAL” interface as its parent as following:

```
public class CustomerXmlDAL : ICustomerDAL
```

Step 3: Register the “Service Class” for “Dependency Injection” in “Startup class for ASP.NET Core 5.0” or “Program class for ASP.NET Core 6.0 and above” by calling any of the 3 register methods => “AddSingleton” or “AddTransient” or “AddScoped” based on your requirement, importing “MVCDHProject.Models” namespace:

ASP.NET Core 5.0: Write the below code in “Startup” Class under “ConfigureServices” Method:

```
services.AddSingleton<ICustomerDAL, CustomerXmlDAL>();
```

Or

```
services.AddScoped<ICustomerDAL, CustomerXmlDAL>();
```

Or

```
services.AddTransient<ICustomerDAL, CustomerXmlDAL>();
```

ASP.NET Core 6.0 or above: Write the below code in “**Program**” class just **above** the statement =>

```
var app = builder.Build();
```

```
builder.Services.AddSingleton<ICustomerDAL, CustomerXmlDAL>();
```

Or

```
builder.Services.AddScoped<ICustomerDAL, CustomerXmlDAL>();
```

Or

```
builder.Services.AddTransient<ICustomerDAL, CustomerXmlDAL>();
```

Note: for our application what we have been developing, “**AddScoped**” will be the best option because in **Insert**, **Update** and **Delete** we have multiple **methods** getting executed in **each request**.

Step 4: Now go to “**CustomerController**” class and in **top** of the **class** we have created **instance** of “**CustomerXmlDAL**” class, **delete it** and write the below code over there:

```
private readonly ICustomerDAL obj;
public CustomerController(ICustomerDAL obj)
{
    this.obj = obj;
}
```

Note: after doing all the above changes run the **Project** again and check the **output** of all the **Action Methods** which executes **as is**. Now we can start implementing the **DAL Class logic targeting** different **Data Sources**.

Implementing DAL Class targeting SQL Server

Now let's implement a new **DAL Class** targeting **SQL Server**, and to do that let's use **Entity Framework Core** which is an extension to **Entity Framework**.

Entity Framework Core:

- This is the new version of **Entity Framework** after **EF 6.x**.
- It is **open-source**, **lightweight**, **extensible** and **cross-platform** version of **Entity Framework** data access technology.
- **Entity Framework** is an **Object/Relational Mapping (O/RM) Framework**.
- It is an **enhancement** to older **ADO.NET** that gives developers an **automated mechanism** for **accessing** and **storing** the data in **Database**.
- **EF Core** is intended to be used with **.NET Core** applications. However, it can also be used with **standard .NET 4.5+ Framework-based** applications.
- **EF Core** is a complete **re-write**, compared to **EF 6** but maximum all options of **EF 6** are available in **EF Core**.

EF Core Development Approaches: EF Core supports only 2 development approaches.

1. **Code-First**
2. **Database-First**

- EF Core mainly targets **Code-First** approach and provides very limited support for **Database-First** approach because the visual designer for **DB Modeling** is not supported.
- In **Code-First** approach, EF Core API creates **Database** and **Tables** using **migrations**, based on the **conventions** and **configuration** provided in our **domain** classes. This approach is useful in **Domain Driven Design (DDD)**.
- In **Database-First** approach, EF Core API creates **Domain** and **Context** classes based on our existing **Database** using **EF Core** commands. This has limited support in **EF Core** as it doesn't have **Visual Designer** or **Wizard**.

EF Core Vs EF 6: EF Core is a new and improved version of Entity Framework for .NET Core applications. EF Core continues to support the following **features** and **concepts**, same as EF 6:

1. **DbContext** and **DbSet**
2. Data Model
3. Querying using Linq-to-Entities
4. Change Tracking
5. Save Changes
6. Migrations

EF Core will include most of the features of EF 6 gradually. However, there are some features of EF 6 which are **not supported** in EF Core such as:

1. EDMX/Graphical Visualization of Model
2. Entity Data Model Wizard (for DB-First approach)
3. Automated Migration
4. Inheritance Strategies: Table per type (TPT) & Table per concrete class (TPC)
5. Many-to-Many without join entity
6. Lazy loading of related data
7. Stored procedure mapping with **DbContext** for **CUD** operation

EF Core includes the following new features which are not supported in EF 6:

1. Easy relationship configuration
2. Batch **INSERT**, **UPDATE**, and **DELETE** operations
3. In-memory provider for testing
4. Support for “**IoC**” (Inversion of Control)
5. Unique constraints
6. Shadow properties
7. Alternate keys
8. Global query filter
9. Field mapping
10. **DbContext** pooling and patterns for handling disconnected entity graphs

EF Core Database Providers (Libraries): Entity Framework Core uses provider model to access many different Database's. Entity Framework Core provides access too many Databases through plug-in libraries called **Database Providers** and we need to install those providers as **NuGet** packages. List of available providers can be found at the following location: <https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli>

To work with Entity Framework Core in our applications we need to install the below 2 packages from NuGet:

- Microsoft.EntityFrameworkCore.Tools
- Microsoft.EntityFrameworkCore.SqlServer

Microsoft.EntityFrameworkCore.Tools: contains a set of **types** and **tools** for **Scaffolding**, using which we can perform **migrations**. This is common for any **Database** we want to work with like **SQL Server** or **Oracle** or **My SQL** and this package internally depends on another **package**: **Microsoft.EntityFrameworkCore.Design**.

- **Microsoft.EntityFrameworkCore.Design:** contains shared design-time components for **Entity Framework Core Tools**.

Microsoft.EntityFrameworkCore.SqlServer: contains a set of **types** which are required to work with **SQL Server Database** and this package internally depends on 2 other **packages**: **Microsoft.EntityFrameworkCore.Relational** and **Microsoft.Data.SqlClient**.

- **Microsoft.EntityFrameworkCore.Relational:** contains all the types that are required in **common** to work with any **Relational Database**.
 - **Microsoft.Data.SqlClient:** contains all the types which are required to work with **SQL Server Database**.
-

To work with **SQL Server** using **Entity Framework Core** in our existing “**MVCDHProject**” application, perform the below actions:

Step 1: Install the above 2 specified **Nuget** packages.

Step 2: Go to “**Customer.cs**” file and apply “**Data Annotations**” on it by importing the required namespaces:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
public class Customer
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Custid { get; set; }

    [MaxLength(100)]
    [Column(TypeName = "Varchar")]
    public string Name { get; set; }

    [Column(TypeName = "Money")]
    public decimal? Balance { get; set; }

    [MaxLength(100)]
    [Column(TypeName = "Varchar")]
    public string City { get; set; }

    public bool Status { get; set; }
}
```

Step 3: Open “**appSettings.json**” file and write the “**ConnectionString**” for connecting to **SQL Server** just below the (“**AllowedHosts**”: “*****”) statement, that should now look as following:

```
{
```

```

    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "ConStr": "Data Source=Server;User Id=Sa;Password=123;Database=MvcCoreDB;TrustServerCertificate=True"
    }
}

```

Step 4: Now add a new class in to the **Models** folder with the name “**MVCCoreDbContext**” and this class is our “**DbContext**” and under this class write the below code:

```

using Microsoft.EntityFrameworkCore;
public class MVCCoreDbContext : DbContext
{
    public MVCCoreDbContext(DbContextOptions options) : base(options)
    {
    }
    public DbSet<Customer> Customers { get; set; }
}

```

Note: the above class “**MVCCoreDbContext**” is not specifically configured to work with **SQL Server Database**, but was designed **generic** to work with any **Relational Database**, so this is also loosely coupled with the **Database** we want to work with. We need to inject the information of **Database** we want to work with thru the **Injector** class i.e., either **Startup** in **ASP.NET Core 5.0** or **Program** in **ASP.NET Core 6.0** and **above**.

Step 5: Because “**MVCCoreDbContext**” class is loosely coupled we need to register this class in **Startup Class** for **ASP.NET Core 5.0** and **Program Class** for **ASP.NET Core 6.0 and above**, to tell whether it has to connect with **SQL Server** or **SQL Lite** or **My SQL** or **Oracle** or **Postgre SQL** or **Cosmos** Database’s, so that the **Dependency Injection Container** will tell “**MVCCoreDbContext**” class to which **Database** it has to connect with, in **runtime**. To register the **Context** class, we need to use “**AddDbContext**” method and to do that write the below code:

ASP.NET Core 5.0: go to “**ConfigureServices**” method of “**Startup**” class, import “**Microsoft.EntityFrameworkCore**” namespace and write the below code:

```

services.AddDbContext<MVCCoreDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("ConStr")));

```

ASP.NET Core 6.0 and above: go to “**Program**” class, import “**Microsoft.EntityFrameworkCore**” namespace and write the below code just **above** the statement => **var app = builder.Build();**

```

builder.Services.AddDbContext<MVCCoreDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("ConStr")));

```

Note: “`UseSqlServer`” is an extension method under “`DbContextOptionsBuilder`” class which is added on install of “`Microsoft.EntityFrameworkCore.SqlServer`” package; same as this when we install the “`Oracle Provider`” package it will add another extension method with the name “`UseOracle`” and so on.

Step 6: Add a class in `Model`’s folder naming it “`CustomerSqlDAL`” and implement the interface “`ICustomerDAL`” and provide implementation to all the `abstract methods of interface`, targeting `SQL Server Database`.

```
public class CustomerSqlDAL : ICustomerDAL
{
    private readonly MVCCoreDbContext context;
    public CustomerSqlDAL(MVCCoreDbContext context)
    {
        this.context = context;
    }
    public List<Customer> Customers_Select()
    {
        var customers = context.Customers.Where(C => C.Status == true).ToList();
        return customers;
    }
    public Customer Customer_Select(int Custid)
    {
        return context.Customers.Find(Custid);
    }
    public void Customer_Insert(Customer customer)
    {
        context.Customers.Add(customer);
        context.SaveChanges();
    }
    public void Customer_Update(Customer customer)
    {
        customer.Status = true;
        context.Update(customer);
        context.SaveChanges();
    }
    public void Customer_Delete(int Custid)
    {
        Customer customer = context.Customers.Find(Custid);
        customer.Status = false;
        context.SaveChanges();
    }
}
```

Note: in the above class we are using `Dependency Injection` to inject the `Context` class object thru the `Constructor`.

Step 7: We are using the `Code First Approach` in our application and we are not provided with `Automatic Migrations` in `EF Core`, so we need to run few `Scaffolding Commands` to create the `Database` and `Tables` and to do that open “`Package Manager Console`” and run the below commands 1 after the other.

PM> `Add-Migration InitialMigration`

The above command will create a folder under the project with the name “**Migrations**” and in the folder it will create a file with the name “**TimeStamp_InitialMigration.cs**” and in this file we find “**Create Table**” and other **SQL Statements** which must be executed on the **Database**, and to execute them run the below command:

PM> Update-Database //This command will create the Database and Table on SQL Server.

Step 8: Now go to **Startup Class** for **ASP.NET Core 5.0** or **Program Class** for **ASP.NET Core 6.0 and above**, and change the “**Service Class Name**” in the method where we registered the interface i.e., **Service Class Name** “**CustomerXmIDAL**” should be changed to “**CustomerSqlDAL**”, as below:

ASP.NET Core 5.0:

```
services.AddSingleton<ICustomerDAL, CustomerSqlDAL>();  
Or  
services.AddTransient<ICustomerDAL, CustomerSqlDAL>();  
Or  
services.AddScoped<ICustomerDAL, CustomerSqlDAL>(); //Preferred
```

ASP.NET Core 6.0:

```
builder.services.AddSingleton<ICustomerDAL, CustomerSqlDAL>();  
Or  
builder.services.AddTransient<ICustomerDAL, CustomerSqlDAL>();  
Or  
builder.services.AddScoped<ICustomerDAL, CustomerSqlDAL>(); //Preferred
```

For our application “**AddScoped**” is the right option because we are dealing with **Database Connections**. Now run the “**CustomerController**” and invoke the **Action Methods** to perform **CRUD Operations**, and the advantage here is the same **Controller** and **Views** can be used now also without **re-writing** them again.

Migration in Entity Framework Core: Migration is a feature which keeps the **Database schema** in **sync** with our **Entity Classes** by preserving data. **EF Core API** builds the **EF Core Model** from the **Domain (Entity) Classes** and **EF Core Migrations** will **create or update** the **Database schema** based on **EF Core Model**. Whenever we change the **Domain Classes**, you need to run **Migration** to keep the **Database schema** up to date. **EF Core Migrations** are a set of commands that we can execute in “**NuGet Package Manager Console**” or in “**Dotnet Command Line Interface**”.

PMC Command	Dotnet CLI Command
Add-Migration <migration name>	Add <migration name>
<u>Usage:</u> Creates a migration by adding a migration snapshot.	
Remove-Migration	Remove
<u>Usage:</u> Removes the last migration snapshot.	
Update-Database	Update
<u>Usage:</u> Updates the database schema based on the last migration snapshot.	
Script-Migration	Script
<u>Usage:</u> Generates a SQL script by using all the migration snapshots.	

Adding a Migration: At the very first time, when we define the initial **Domain Classes** and there is no **Database** for our **Application** then we need to create our **first migration**, to **create the Database** as well as whenever we change our **Domain Classes**, we need to create a **new migration** to **update the Database**.

Package Manager Console: PM> Add-Migration InitialMigration

.NET CLI: <drive>:<Personal Folder>\MVCDHProject\MVCDHProject> dotnet ef migrations Add InitialMigration

Note: in the above statements, “InitialMigration” is the name of **Migration** which will create a **Migration** folder with 2 files in it:

1. **<timestamp> <InitialMigration>.cs:** this file includes migration operations in the **Up()** and **Down()** methods. The **Up()** method includes code for creating **Database Objects** and **Down()** method includes code for dropping **Database objects**.
 2. **<contextclassname>ModelSnapshot.cs:** this is a **snapshot** of your current **model** which is used to determine what is changed when creating the next **migration**.
-

Updating the Database: we use this command to create or update **Database** schema based on the **Migration** we have created above.

Package Manager Console: PM> Update-Database

.NET CLI: <drive>:<Personal Folder>\MVCDHProject\MVCDHProject> dotnet ef database Update

Note: This command will create the **Database** (if it is not existing) based on the **Context**, **Domain Classes** and the **Migration Snapshot**, which is created using the “add-migration” or “add” commands. If this is the **first migration**, it will also create a **Table** with the name “**__EFMigrationsHistory**”, which will store the names of all migrations that are executed till now which are applied to the **Database**.

Removing a Migration: we can remove the **last migration**, provided it is **not applied** to the **Database**. Use the following remove commands to remove the last created migration files which will revert the model snapshot.

Package Manager Console: PM> Remove-Migration

.NET CLI: <drive>:<Personal Folder>\MVCDHProject\MVCDHProject> dotnet ef migrations Remove

Note: if the migration is already applied to **Database**, then it will throw an exception and displays below error message:

The migration “<Migration Name>” has already been applied to the database. Revert it and try again. If the migration has been applied to other databases, consider reverting its changes using a new migration instead.

Reverting a Migration: suppose we changed our domain classes and created a second migration with the name “**NewMigration1**” using the “add-migration” command and applied that migration to **Database** using “**Update Command**” but, for some reason, we want to **revert** or **rollback** the Database to its previous state then we need to use the “**Update-Database <migration name>**” command to revert the Database to the specified previous migration snapshot.

Package Manager Console: PM> Update-Database InitialMigration

.NET CLI: <drive>:\<Personal Folder>\MVCDHProject\MVCDHProject> dotnet ef database Update InitialMigration

The above command will revert the **Database** based on specified migration named “InitialMigration” and removes all the changes applied for second migration named “NewMigration1”. This will also **remove** the entry from the “__EFMigrationsHistory” table in the **Database**. But this will not remove the migration file related to “NewMigration1”, so we need to use the **Remove** command and remove them from the project.

Generating a SQL Script: use this command to generate a SQL script for the database.

Package Manager Console: PM> script-migration

.NET CLI: <drive>:\<Personal Folder>\MVCDHProject\MVCDHProject> dotnet ef migrations script

The above command will include a script for all the migrations by default, and we can specify a range of migrations also by using the “-to” and “-from” options.

Help Commands: Package Manager Console provides us a command called “get-help” using which we can get help about any required topic.

Package Manager Console: PM> Get-Help EntityFramework

The above command will provide information about the “EF Core Commands” and same as this we can also use “Get-Help” on each “EFCORE” Commands as following:

Package Manager Console: PM> Get-Help Add-Migration

To try the above commands, do the following:

Step 1: Go to **Customer** class and add a **new property** in the class as below:

```
public string State { get; set; }
```

Step 2: Open **Package Manager Console** and create a **new migration** as below:

```
PM> Add-Migration Update1
```

The above action will create a new migration file under “**Migrations**” folder with the name “<time stamp>_Update1.cs” which contains an “Up” method for adding the new column and “Down” method for dropping the column. The Snapshot file will contain the modifications that are made.

Step 3: Go to **Customer** class again and add a **new property** as below:

```
public string Country { get; set; }
```

Step 4: Open **PMC** and create a **new migration** as below:

```
PM> Add-Migration Update2
```

The above action will create a new migration file under “**Migrations**” folder with the name “<time stamp>_Update2.cs” which contains an “Up” method for adding the new column and “Down” method for dropping the column. The Snapshot file will contain the modifications that are made.

Step 5: we can now use the **Remove Migration** command to remove the **last created migration** file as following at **Package Manager Console**:

```
PM> Remove-Migration
```

The above action will remove the last migration file i.e., “**Update2**” and this action will also update the **Snapshot** file.

Step 6: let us apply “**Update1**” migration on the **Database** and to do that use the below command at **Package Manager Console**:

```
PM> Update-Database
```

The above action will update the **last migration** file i.e., “**Update1**” to the **Database** but the property “**Country**” is not removed from the “**Customer**” class, which should be explicitly removed by us or we can even create a new “**Migration**” for that property update.

Step 7: Open **Package Manager Console** and create a **new migration** as below:

```
PM> Add-Migration Update2
```

The above action will create a new migration file under “**Migrations**” folder with the name “**<time stamp>_Update2.cs**” which contains an “**Up**” method for adding the new column and “**Down**” method for dropping the column. The Snapshot file will contain the modifications that are made.

Step 8: Go to **Customer** class again and add a **new property** as below:

```
public string Continent { get; set; }
```

Step 9: Open **Package Manager Console** and create a new Migration as below:

```
PM> Add-Migration Update3
```

The above action will create a new migration file under “**Migrations**” folder with the name “**<time stamp>_Update3.cs**” which contains an “**Up**” method for adding the new column and “**Down**” method for dropping the column. The Snapshot file will contain the modifications that are made.

Step 10: Open **Package Manager Console** and **update** the **Database** as below:

```
PM> Update-Database
```

The above action will update both **Migrations** to the Database i.e., “**Update2**” and “**Update3**” also, and we can verify this by connecting to SQL Server and querying on the table: “**__EFMigrationsHistory**” which will show the history of all migrations we have created till now.

Note: now if we try to use the “**Remove-Migration**” command it will throw an **error** as all the **migrations** are **updated** to the **Database** and to test that go to **Package Manager Console** and use the below command:

```
PM> Remove-Migration
```

Rollback Migration: If we want to bring our **Database Schema**, back to its any state i.e., to any of the **Migrations** we applied, we can revert to a migration and to do that, we need to use “**Update-Database**” command with the “**Migration Name**” (you can get the name of **Migration** to where we want to rollback, from the

“`__EFMigrationsHistory`” table) we want to revert, and to test that go to **Package Manager Console** and use the below command:

PM> `Update-Database <time stamp value>_InitialMigration`

The above action will **roll back** to “`InitialMigration`” and updates the table “`__EFMigrationsHistory`” but it will not remove the **Migration Files** under the project and to remove them we need to use “**Remove-Migration**” command for **3 times** because each time it remove only **1 migration**.

Note: these **actions** will not **remove** the properties we added in our **Customer** class, so go there and explicitly **delete** them also.

Data Seeding: this is a process of populating **Database-Tables** with some initial data and to do that we need to override “`OnModelCreating`” method in our “`DataContext`” class i.e., “`MVCCoreDbContext`”, and to test this process write the below code in the class:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>().HasData(
        new Customer { Custid = 101, Name = "Sai", Balance = 50000.00m, City = "Delhi", Status = true },
        new Customer { Custid = 102, Name = "Sonia", Balance = 40000.00m, City = "Mumbai", Status = true },
        new Customer { Custid = 103, Name = "Pankaj", Balance = 30000.00m, City = "Chennai", Status = true },
        new Customer { Custid = 104, Name = "Samuels", Balance = 25000.00m, City = "Bengaluru", Status = true }
    );
}
```

Note: generally, **seed data** is performed in the initial stages only i.e., the first time we create the **Database** then only we will perform data seeding (preferably in initial migration).

Step 2: go to **Package Manager Console** and create a **new migration** as below:

PM> `Add-Migration SeedData`

Step 3: update the **Database** from **Package Manager Console** using the below statement:

PM> `Update-Database`

Note: This action will **update** the migration “`SeedData`” to the **Database** and inserts the **4 records** into **Table** but before updating the **migration**, go to the **Database** and delete if there are any existing records in the table.

Handling Errors in ASP.NET Core MVC Applications

In **Web Applications** we generally come across 2 different types of **Errors**:

1. **Http Status Code Based Errors (4XX)** => Client Errors
2. **Exceptions (5XX)** => Server Errors

Handling Client Errors: These **errors** occur if at all the request contains **bad syntax** or cannot be **fulfilled**. This category of **status codes** is intended for situations in which the error seems to have been caused by **client**. For example, if we try to send a request for a page which is not existing then it is “**404; Not Found Error**”. To handle these **errors**, **MVC Core** provides us **3 Middle Ware** components and they are:

1. UseStatusCodePages
2. UseStatusCodePagesWithRedirects
3. UseStatusCodePagesWithReExecute

Note: to test these, first change the “ASPNETCORE_ENVIRONMENT” setting to “Production” in “launchSettings.json” under “IIS Express” and “Kestrel Web Server” profile because by default it is “Development”.

To use these **Middleware’s** first let’s try to access a page which is not existing in the site, for example: <http://localhost:port/Customer/GetCustomer> this will display an **error page** to the user which we generally see in real-time when a resource was not found.

UseStatusCodePage: this Middleware component adds a “StatusCodePage” with a default response that displays the “Status Code” and “Status Message”. To use the above **Middleware**, do the following actions in “Startup.cs” file for ASP.NET Core 5.0 and “Program.cs” file in ASP.NET Core 6.0 and above:

ASP.NET Core 5.0: go to “Startup.cs” file and change the existing code that is present under “Configure” method of the class as below. Old code you find in the method will be as following:

```
if (env.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
}
else {
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

Change it as below:
if (env.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
}
else {
    app.UseStatusCodePages();
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```

ASP.NET Core 6.0 and above: go to “Program.cs” file and change the existing code that is present under the class as below. Old code you find over there will be as following:

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

Change it as below:
if (!app.Environment.IsDevelopment())
{
    app.UseStatusCodePages();
```

```

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

```

Now try to access a page which is not existing again, but now we see the **Error Page** displaying the **Status Code** and **Status Message** as following: “**Status Code: 404; Not Found**”.

UseStatusCodePagesWithRedirects: this **Middleware** specifies that the response should be handled by redirecting to the given location **URL** template which may include “{0}” => **Place Holder** for **Status Code**. To use this **Middleware**, do the following:

Step 1: Add a controller in our “**MVCDHProject**” naming it as “**ErrorController**” and write the below code in it by deleting the existing code in the class:

```

[Route("ClientError/{StatusCode}")]
public IActionResult ClientErrorHandler(int StatusCode)
{
    switch(StatusCode)
    {
        case 400:
            ViewBag.ErrorTitle = "Bad Request";
            ViewBag.ErrorMessage = "The server can't return a response due to an error on the client's end.";
            break;
        case 401:
            ViewBag.ErrorTitle = "Unauthorized or Authorization Required";
            ViewBag.ErrorMessage = "Returned by server when the target resource lacks authentication credentials.";
            break;
        case 402:
            ViewBag.ErrorTitle = "Payment Required";
            ViewBag.ErrorMessage = "Processing the request is not possible due to lack of required funds.";
            break;
        case 403:
            ViewBag.ErrorTitle = "Forbidden";
            ViewBag.ErrorMessage = "You are attempting to access the resource that you don't have permission to view.";
            break;
        case 404:
            ViewBag.ErrorTitle = "Not Found";
            ViewBag.ErrorMessage = "The requested resource does not exist, and server does not know if it ever existed.";
            break;
        case 405:
            ViewBag.ErrorTitle = "Method Not Allowed";
            ViewBag.ErrorMessage = "Hosting server supports the method received, but the target resource doesn't.";
            break;
        default:
            ViewBag.ErrorTitle = "Client Error Occured";
            ViewBag.ErrorMessage = "There is a Client-Error in the page, re-check the input you supplied.";
            break;
    }
}

```

```

    }
    return View("ClientErrorView");
}

```

Step 2: Add a new View in the **Shared** folder of our project naming it as “**ClientErrorView**” and write the below code in it:

```

<h2 class="alert-danger">
    Client Error Page - @ViewBag.ErrorTitle
</h2>
<p style="text-align:justify;font-size:large">
    @ViewBag.ErrorMessage
</p>

```

Step 3: Go to “**Startup.cs**” or “**Program.cs**” file (based on the version you are targeting), **comment** the method “**“UseStatusCodePages();”** and in that place call the method “**“UseStatusCodePagesWithRedirects”** as following:

```
app.UseStatusCodePagesWithRedirects("/ClientError/{0}");
```

In this case when we get any **status code-based error or client error** it will redirect to “**ErrorController**” passing the **“StatusCode”** as a parameter and invokes the method **“ClientErrorHandler”** because we have defined a route to **“UseStatusCodePagesWithRedirects”** method as “[Route(“ClientError/{StatusCode}”)]” and now **“ClientErrorHandler”** method will launch the view i.e., **“ClientErrorView”** and displays the details of error.

Note: whenever a client error occurs “**UseStatusCodePagesWithRedirects**” Middleware will transfer the control to browser with a redirection response **(302)** and then browser will send a **new request** to the action method with the route **“ClientError”** and passes the status code **404 (in our case now)** so that the URL in address bar gets updated to <http://localhost:port/ClientError/404>. The drawback in this case is there is an **extra round trip** between the **Browser** and **Web Server**.

UseStatusCodePagesWithReExecute: this **Middleware** specifies that the response body should be generated by re-executing the request pipeline using an alternate path which may include “**{0}**” Placeholder for **Status Code**. We can overcome the problem of **“UseStatusCodePagesWithRedirects”** with **“UseStatusCodePagesWithReExecute”** Middleware, because in this case it will not perform a re-direction but transfers the control from 1 page to another page directly on the server itself, so the **URL** in browsers address bar will not be updated i.e., it will be still showing the original requested URL only. To test this, go to “**Startup.cs**” or “**Program.cs**” file (based on the version you are targeting), **comment** the method call **“UseStatusCodePagesWithRedirects”** and call the new method **“UseStatusCodePagesWithReExecute”** as following:

```
app.UseStatusCodePagesWithReExecute("/ClientError/{0});
```

Handling Server Errors: to handle **server errors** we need to configure the middleware **“UseExceptionHandler”** and specify the address where it has to be redirected when there is a server error. Currently in an **“MVC Project”** it is already configured, so if we observe the code in **Configure** method of **Startup** class (**ASP.NET Core 5.0**) and **Program** class (**ASP.NET Core 6.0 & above**) we notice the statement: “**app.UseExceptionHandler("/Home/Error");**”, and right now this is already configured to invoke the **“Error”** action method of **“Home”** Controller, so whenever there is a server error it will redirect to the **Home Controller - Error Action** method. To test this, go to **“CustomerSqlDAL”** class and re-write **“Customer_Select”** method as below:

```

public Customer Customer_Select(int Custid) {
    Customer customer = context.Customers.Find(Custid);
    if (customer == null) {
        throw new Exception("No customer exist's with given Custid.");
    }
    return customer;
}

```

Now send a request for DisplayCustomer action method with Customer-Id which is not existing as below:

<https://localhost:port/Customer/DisplayCustomer?Custid=110>

Note: the above request will now cause a **Server Error (Exception)** and redirects to **Home Controller – Error Action** method and displays the corresponding **View** over there. If we want to handle the error on our own without using the **pre-defined** code, do the following:

Step 1: Go to “**Startup.cs**” or “**Program.cs**” file (**based on the version you are targeting**), change the code which is calling the method “**UseExceptionHandler**” as below, so that whenever there is an error without going to **Home Controller - Error Action** method, it will redirect to the **Action Method** with the route name as “**ServerError**”:

```
app.UseExceptionHandler("/Home/Error"); => app.UseExceptionHandler("/ServerError");
```

Step 2: Go to “**ErrorController**” class and add the below action method in that class by importing the namespace “**Microsoft.AspNetCore.Diagnostics**”:

```

[Route("ServerError")]
public IActionResult ServerErrorHandler() {
    var ExceptionDetails = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
    ViewBag.ErrorTitle = ExceptionDetails.Error.GetType().Name;
    ViewBag.Path = ExceptionDetails.Path;
    ViewBag.ErrorMessage = ExceptionDetails.Error.Message;
    return View("ServerErrorView");
}

```

Note: The “**ServerErrorHandler**” action method is defined with the route as “**ServerError**” so it gets invoked when there is a server error as we configured it in **Startup Class (ASP.NET Core 5.0)** and **Program Class (ASP.NET Core 6.0 and above)**, which launches the view “**ServerErrorView**” by passing the details of the **error** thru **ViewBag**.

Step 3: Add a **View** in **Shared** folder of our project, naming it as “**ServerErrorView.cshtml**” and write the below code in it:

```

<h2 class="align-content-xl-center alert-danger">
    Server Error Page - @ViewBag.ErrorTitle
</h2>
<h4>
    There is a error at @ViewBag.Path, and the details of the error are:
</h4>
<p style="text-align:justify;font-size:large;color:red">
    @ViewBag.ErrorMessage
</p>

```

ASP.NET Core Identity Framework

It is an API that supports UI Login functionality by managing users, passwords, profile data, roles, tokens, email confirmation, and more. Users can create an account with the login information stored in Identity or they can use external login providers that include Facebook, Google, Microsoft, and Twitter Accounts. Identity is typically configured using an SQL Server Database to store usernames, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

To work with ASP.NET Core Identity, do the following:

Step 1: Install “Microsoft.AspNetCore.Identity.EntityFrameworkCore” package by using NuGet Package Manager.

Step 2: Go to our context class i.e., “MVCCoreDbContext” and change the parent class of it from “DbContext” to “IdentityDbContext” by importing “Microsoft.AspNetCore.Identity.EntityFrameworkCore” namespace and this “IdentityDbContext” class internally inherits from “DbContext” class anyhow.

Step 3: In our “DbContext” class i.e., “MVCCoreDbContext” with-in the existing “OnModelCreating” method which we have overridden earlier, write the below statement in top of the method, so that the base class i.e., “IdentityDbContext”, “OnModelCreating” method also gets executed, and we need to do this call because there is some logic implemented in the parent class for setting “IdentityUserLogin” model’s key property:

```
base.OnModelCreating(modelBuilder);
```

Step 4: Register “IdentityFramework” in our application by importing “Microsoft.AspNetCore.Identity” namespace:

ASP.NET Core 5.0: Go to “Startup.cs” file, and write the below code under “ConfigureServices” method:

```
services.AddIdentity<IdentityUser, IdentityRole>().AddEntityFrameworkStores<MVCCoreDbContext>();
```

ASP.NET Core 6.0 and above: Go to “Program.cs” file, and write the below code just above the statement:

```
    "var app = builder.Build();"
```

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>().AddEntityFrameworkStores<MVCCoreDbContext>();
```

Step 5: Add Authentication Middleware to request pipeline using the below statement:

```
    app.UseAuthentication();
```

ASP.NET Core 5.0: Go to “Startup.cs” file, and write the above statement in “Configure” method between “UseRouting” and “UseAuthorization” Middleware’s.

ASP.NET Core 6.0 and above: Go to “Program.cs” file, and write the above statement between “UseRouting” and “UseAuthorization” Middleware’s:

Step 6: Now open Package Manager Console and create a new Migration as below:

```
PM> Add-Migration AddIdentity
```

Step 7: Update the new migration i.e., “AddIdentity” as below, so that all the required Tables gets created on our Database i.e., “MVCCoreDb”:

```
PM> Update-Database
```

Step 8: Now under the **Models** folder define the below classes:

```
using System.ComponentModel.DataAnnotations;
public class UserModel
{
    [Required]
    public string Name { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm Password")]
    [Compare("Password", ErrorMessage = "Confirm password should match with password.")]
    public string ConfirmPassword { get; set; }

    [Required]
    [EmailAddress]
    [Display(Name = "Email Id")]
    public string Email { get; set; }

    [Required]
    [RegularExpression("[6-9]\\d{9}", ErrorMessage = "Mobile No. Is Invalid")]
    public string Mobile { get; set; }
}

using System.ComponentModel.DataAnnotations;
public class LoginModel
{
    [Required]
    public string Name { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember Me")]
    public bool RememberMe { get; set; }
}
```

Step 9: Now under the **Controllers** folder add a new controller with the name "**AccountController**", delete the existing **Index Action** method in it and write the below code:

```
using MVCDHProject.Models;
using Microsoft.AspNetCore.Identity;
```

```

public class AccountController : Controller
{
    private readonly UserManager<IdentityUser> userManager;
    private readonly SignInManager<IdentityUser> signInManager;
    public AccountController(UserManager<IdentityUser> userManager, SignInManager<IdentityUser> signInManager)
    {
        this.userManager = userManager;
        this.signInManager = signInManager;
    }

    public IActionResult Register()
    {
        return View();
    }
    [HttpPost]
    public async Task<IActionResult> Register(UserModel userModel)
    {
        if (ModelState.IsValid)
        {
            //IdentityUser represents a new user with a given set of attributes
            IdentityUser identityUser = new IdentityUser { UserName = userModel.Name, Email = userModel.Email,
                PhoneNumber = userModel.Mobile };

            //Creates a new user and returns a result which tells about success or failure
            var result = await userManager.CreateAsync(identityUser, userModel.Password);
            if (result.Succeeded)
            {
                //Performing a Sign-In into the application
                await signInManager.SignInAsync(identityUser, false);
                return RedirectToAction("Index", "Home");
            }
            else
            {
                foreach (var Error in result.Errors)
                {
                    //Displaying error details to the user
                    ModelState.AddModelError("", Error.Description);
                }
            }
        }
        return View(userModel);
    }
}

```

UserManager class provides options for managing users in a persistence store.

IdentityUser class represents a user in the identity system.

SignInManager class provides options for user sign in.

Adding a Register View: right click on the `Register` action method in the “`AccountController`” class, select “Add View”, in the window opened select “Razor View”, click “Add” button, in the window opened, select `Template` as “Create”, `Model Class` as “`UserModel (MVCDHProject.Models)`” and click on the “Add” button which will create a `View` with the name “`Register.cshtml`”. After the View is created, make any changes in the design (if required) and then run the view to register new users.

Note: When we want to create a new user account a set of rules are pre-defined on `Username` and `Password`, under 2 classes: “`UserOptions`” and “`PasswordOptions`”.

“`UserOptions`” class defines options for `user validations` and in this class, there is a property “`AllowedUserNameCharacters`” which contains the list of `allowed characters` for the “`Username`” and in this the code is as following:

```
public class UserOptions
{
    public string AllowedUserNameCharacters { get; set; } =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    public bool RequireUniqueEmail { get; set; }
}
```

“`PasswordOptions`” class defines options for `password validations` and in this class, there are properties like `RequiredLength`, `RequiredDigit`, `RequiredLowerCase`, `RequiredUpperCase`, etc. to validate the `password`, and in this class the code is as following:

```
public class PasswordOptions
{
    public int RequiredLength { get; set; } = 6;
    public int RequiredUniqueChars { get; set; } = 1;
    public bool RequireNonAlphanumeric { get; set; } = true;
    public bool RequireLowercase { get; set; } = true;
    public bool RequireUppercase { get; set; } = true;
    public bool RequireDigit { get; set; } = true;
}
```

We can override these options as per our requirements and we need to do that in “`Startup.cs`” file for `ASP.NET Core 5.0` and “`Program.cs`” file for `ASP.NET Core 6.0` and `above`. For example, if we want to set the “`RequiredLength`” as `8` characters and “`RequireDigit`” as `false` do the following:

ASP.NET Core 5.0: go to “`Startup.cs`” file and re-write the “`Services.AddIdentity`” method as following:

Old Code:

```
services.AddIdentity<IdentityUser, IdentityRole>().AddEntityFrameworkStores<MVCCoreDbContext>();
```

New Code:

```
services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
```

```
options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCDbContext>();
```

ASP.NET Core 6.0 and above: go to “[Program.cs](#)” file and re-write the “`builder.Services.AddIdentity`” method as following:

Old Code:

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>().AddEntityFrameworkStores<MVCCoreDbContext>();
```

New Code:

```
builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
    options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCCoreDbContext>();
```

Step 10: Implementing “[Login Action](#)”, and to do that go to “[AccountController](#)” class and add the below 2 [Action Methods](#) in the class:

```
public IActionResult Login()
{
    return View();
}
[HttpPost]
public async Task<IActionResult> Login(LoginModel loginModel)
{
    if(ModelState.IsValid)
    {
        var result = await signInManager.PasswordSignInAsync(loginModel.Name, loginModel.Password,
                                                               loginModel.RememberMe, false);
        if(result.Succeeded)
        {
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", "Invalid login credentials.");
        }
    }
    return View(loginModel);
}
```

Add a [View](#) to [Login Action](#) method and to do that, right click on [Login](#) method in the [Controller](#) class, select “[Add View](#)”, in the window opened select “[Razor View](#)”, click on “[Add](#)” button, in the window opened select [Template as “Create”](#), [Model Class as “LoginModel \(MVCDHProject.Models\)”](#) and click on the “[Add](#)” button which will create a [View](#) with the name “[Login.cshtml](#)”. Make any necessary changes to the [View](#) (if required) and run it to [Login](#) into the application with the [user accounts](#) you have [registered](#).

Providing Login, Register and Logout links: Let's provide links for **Login** and **Register** if the user is not **signed-in** and if **signed-in**, lets display **Username** and **Logout** links on the top of every page and to do that do the following:

i. Open “_ViewImports.cshtml” file that is present in “Views” folder and write the below statement in it:

```
@using Microsoft.AspNetCore.Identity
```

ii. Open “_Layout.cshtml” file that is present in the “Shared” folder of “Views” folder and write the below statement on top of the file to inject “SignInManager”:

```
@inject SignInManager<IdentityUser> signInManager
```

iii. In “_Layout.cshtml” file, we find a “<div>” tag with an un-ordered list containing links for “Home” and “Privacy” Views and with-in that “” add 1 more “” to the existing 2 “'s” to display a link for **Managing Customers**:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-controller="Customer" asp-action="DisplayCustomers">Customers</a>
</li>
```

Now after the “” tag with-in the same “<div>” tag write the below code to add **Login**, **Logout** and **Register** links on the RHS:

```
<ul class="navbar-nav ml-auto">
  @if (signInManager.IsSignedIn(User))
  {
    <li class="nav-item">
      <a class="nav-link text-dark">@User.Identity.Name</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-controller="Account" asp-action="Logout">Logout</a>
    </li>
  }
  else
  {
    <li class="nav-item">
      <a class="nav-link text-dark" asp-controller="Account" asp-action="Register">Register</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-controller="Account" asp-action="Login">Login</a>
    </li>
  }
</ul>
```

Step 11: Implementing **Logout Action**, and to do that go to “**AccountController**” class and add the below action method over there:

```
public async Task<IActionResult> Logout()
{
```

```
await signInManager.SignOutAsync();
return RedirectToAction("Login");
}
```

Authorization: this refers to the process that determines what a user can do. For example, in a **Library Management Application** an **administrative user** can **create** a document library, **add** documents, **edit** documents, and **delete** them. A **non-administrative user** working with the **library** is only authorized to **read** the documents. **Authorization** is **independent** from **authentication**. However, **authorization** requires an **authentication** mechanism. **Authentication** is the process of **ascertaining** who a user is. Now let's implement **authorization** in our application, so that only **logged-in users** can perform **Edit, Insert** and **Delete** operations on **Customer**, whereas all other **users** can **View** the **Customers** data. To implement this, we have 2 options:

Option 1: **Authorize Attribute** - by using **Authorize** attribute we can specify that this action method is accessible only to **authorized users** and to test that, go to **Customer Controller** class and on top of "**AddCustomer**" (Get Method), "**EditCustomer**" and "**DeleteCustomer**" add the **Authorize** attribute as following by importing "**Microsoft.AspNetCore.Authorization**" namespace:

```
[Authorize]
public ViewResult AddCustomer()
```

```
[Authorize]
public ViewResult EditCustomer(int Custid)
```

```
[Authorize]
public RedirectToActionResult DeleteCustomer(int Custid)
```

Note: Now run the application and try to access the above **3 action methods** without **signing** into the application, which will redirect you to **Login Page**.

Without using **Authorize** attribute on each **action** method we can use it directly on the **Controller class** also, so that all **Action** methods in the class can be accessed only by **Authorizing** and in this case if we want to provide **Anonymous** access to any of the methods in that **Controller** class we need to use "**AllowAnonymous**" attribute on those methods.

To test this comment or delete **Authorize** attribute we used on the **3 action methods** and apply the **Authorize** attribute on **Controller** class as well as on the top of "**DisplayCustomer**" and "**DisplayCustomers**" action methods use "**AllowAnonymous**" attributes as following:

```
[Authorize]
public class CustomerController : Controller
```

```
[AllowAnonymous]
public ViewResult DisplayCustomers()
```

```
[AllowAnonymous]
public ViewResult DisplayCustomer(int Custid)
```

Note: “AllowAnonymous” attribute bypasses all **authorization** statements i.e., if we combine “AllowAnonymous” and “Authorize” attributes, then “Authorize” attribute is ignored. For example, if you apply “AllowAnonymous” at **Controller** level, any “Authorize” attributes on that **Controller’s** action methods will be ignored.

Option 2: if we want to apply **authorization** on all the **controllers** in the **project**, then without applying it on each and every **controller** we can do that in “**Startup.cs**” file for **ASP.NET Core 5.0** and “**Program.cs**” file for **ASP.NET Core 6.0** and **above** and to do that, first import the 2 namespaces: “**Microsoft.AspNetCore.Authorization**” and “**Microsoft.AspNetCore.Mvc.Authorization**”, and re-write the statement “**services.AddControllersWithViews()**” as below:

ASP.NET Core 5.0: go to “**Startup.cs**” file and **re-write** the code of “**AddControllersWithViews**” method that is present under “**ConfigureServices**” method as below:

Old Code:

```
Services.AddControllersWithViews();
```

New Code:

```
services.AddControllersWithViews(configure =>
{
    var policy = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();
    configure.Filters.Add(new AuthorizeFilter(policy));
});
```

ASP.NET Core 6.0 and above: go to “**Program.cs**” file and **re-write** the code of “**AddControllersWithViews**” method that is present above the statement => “**var app = builder.Build();**” as below:

Old Code:

```
builder.Services.AddControllersWithViews();
```

New Code:

```
builder.Services.AddControllersWithViews(configure =>
{
    var policy = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();
    configure.Filters.Add(new AuthorizeFilter(policy));
});
```

With this change, now any **Controller** in the project will not run because we enabled **Authentication** application level i.e., **Home Controller**, **Error Controller** and **Account Controller** also will not run, and to over the problem add “**AllowAnonymous**” attribute on the top of these 3 **Controller** classes by importing “**Microsoft.AspNetCore.Authorization**” namespace:

```
[AllowAnonymous]
public class HomeController : Controller
[AllowAnonymous]
public class ErrorController : Controller
[AllowAnonymous]
public class AccountController : Controller
```

Return URL: after bringing **Authentication** into picture if we try to open any of the secured resource it will first open Login View to Login, but after the Login process is completed it will take us to Index action of Home controller but not to the View we have selected and to overcome this problem, when we are being re-directed to **Login Page** from any secure resource then the “URL” will contain the address of the page to where it has to be redirected after **Login**, and we find it with the Key-Name “**ReturnUrl**”, so we can use this “**Return-URL**” for redirecting back to the original View we requested after a successful **Login** and to do that make the below changes:

Step 1: Go to “**LoginModel.cs**” and add a new property in **LoginModel** class to read the **Return-URL** as below:

```
public string returnUrl { get; set; } = "";
```

Step 2: Go to “**Login.cshtml**” and read the “**ReturnUrl**” which is present in the form of “**Query String**” and to do that, do the following change on the top of the page:

Old Code:

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

New Code:

```
@  
{  
    ViewData["Title"] = "Login";  
    var returnUrl = @Context.Request.Query["ReturnUrl"];  
}
```

Step 3: Now pass the “**returnUrl**” value to **Login’s Post Action Method** as a **route** and to that, change the code of “**<form>**” tag as following:

Old Code:

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

New Code:

```
<form asp-action="Login" asp-route-ReturnUrl="@returnUrl">
```

Step 4: In the **Login’s Post Action Method** of **Account Controller** change the code under “**Succeeded**” condition as below:

Old Code:

```
if (result.Succeeded)  
{  
    return RedirectToAction("Index", "Home");  
}  
else  
{  
    ModelState.AddModelError("", "Invalid login credentials.");  
}
```

New Code:

```
if (result.Succeeded)
{
    if (string.IsNullOrEmpty(loginModel.ReturnUrl))
        return RedirectToAction("Index", "Home");
    else
        return LocalRedirect(loginModel.ReturnUrl);
}
else
{
    ModelState.AddModelError("", "Invalid login credentials.");
}
```

Note: Now run the application and watch the difference.

Email Confirmation: this is quite an important part of the **user registration process**. It allows us to verify the registered **user** is indeed an owner of the provided **email**.

Why email confirmation is important?

Ans: let's imagine a scenario where two users with similar email addresses want to register in our application, for example “Rajann” registers first with “rajan@gmail.com” instead of “rajann@gmail.com” which is his actual address and without email confirmation, this registration will execute successfully. Now, “Rajan” comes to registration page and tries to register with his email “rajan@gmail.com” and our application will return an error that the user with that email is already registered. So, thinking that he already has an account, he just **resets** the **password** and successfully logs in to the application. We can understand with this now where this could lead, and what problems it could cause.

To **overcome** this problem, it is a good practice to ask a user to **confirm** the **email** after **registration** and to implement this in our application do the following changes:

Step 1: Add a new **View** in the **Shared** folder of **Views** folder with the name “**DisplayMessages.cshtml**”, to display any information or error messages to the end users, and write the below code in it deleting the existing content:

```
<h1 class="bg-info text-white">@ TempData["Title"]</h1>
<div class="container">
    <p class="text-justify text-primary">@ TempData["Message"]</p>
</div>
```

Step 2: Go to **Register Post Action Method** in **Account Controller** class and **re-write** the code under the “**Succeeded**” condition as following:

Old Code:

```
if (result.Succeeded)
{
    //Code implemented for an automatic Sign-In after the registration is complete.
    await signInManager.SignInAsync(identityUser, false);
    return RedirectToAction("Index", "Home");
}
```

New Code:

```
if (result.Succeeded)
{
    //Implementing logic for sending a mail to confirm the Email
    var token = await userManager.GenerateEmailConfirmationTokenAsync(identityUser);
    var confirmationUrlLink = Url.Action(
        "ConfirmEmail", "Account", new { UserId = identityUser.Id, Token = token }, Request.Scheme);
    SendMail(identityUser, confirmationUrlLink, "Email Confirmation Link");
    TempData["Title"] = "Email Confirmation Link";
    TempData["Message"] = "A confirm email link has been sent to your registered mail, click on it to confirm.";
    return View("DisplayMessages");
}
```

Step 3: To send an **Email** from our application, first install “**MailKit**” package from **NuGet Package Manager** and then in the **Account Controller** class import the namespaces “**System.Text**”, “**MimeKit**” and “**MailKit.Net.Smtp**”, and add a new method in the class for sending an **Email** as following:

```
public void SendMail(IdentityUser identityUser, string requestLink, string subject)
{
    StringBuilder mailBody = new StringBuilder();
    mailBody.Append("Hello " + identityUser.UserName + "<br /><br />");
    if (subject == "Email Confirmation Link") {
        mailBody.Append("Click on the link below to confirm your email:");
    }
    else if (subject == "Change Password Link") {
        mailBody.Append("Click on the link below to reset your password:");
    }
    mailBody.Append("<br />");
    mailBody.Append(requestLink);
    mailBody.Append("<br /><br /> ");
    mailBody.Append("Regards");
    mailBody.Append("<br /><br />");
    mailBody.Append("Customer Support.");

    BodyBuilder bodyBuilder = new BodyBuilder();
    bodyBuilder.HtmlBody = mailBody.ToString();

    MailboxAddress fromAddress = new MailboxAddress("Customer Support", "<Use your Email Id here>");
    MailboxAddress toAddress = new MailboxAddress(identityUser.UserName, identityUser.Email);

    MimeMessage mailMessage = new MimeMessage();
    mailMessage.From.Add(fromAddress);
    mailMessage.To.Add(toAddress);
    mailMessage.Subject = subject;
    mailMessage.Body = bodyBuilder.ToMessageBody();
```

```

SmtpClient smtpClient = new SmtpClient();
smtpClient.Connect("smtp.gmail.com", 465, true);
smtpClient.Authenticate("<Use your Email Id here>", "<Generate an App Password and use it here>");
smtpClient.Send(mailMessage);
}

```

Note: to send mails from your **Gmail Account** first you need to go to “**Manage your Google Account**” => Select “**Security**” tab => scroll down to “**Signing in to Google**” => under that set the “**2-Step Verification**” property value as “**On**” and **configure** it. Now click on “**App passwords**” option => in the “**Select app**” Dropdown List, choose => **Other (Custom name)** and enter some name in the TextBox over there for example “**TestApp**” and click on **Generate** button which displays a “**16-character Password**”, use this **password** in your application as **password** for the **Email**.

Step 4: To generate token for **Email Confirmation**, **Change Email** and **Reset Password**, etc. we need to register a default **Token Provider** in by calling “**AddDefaultTokenProviders**” method under “**Startup.cs**” file in **ASP.NET Core 5.0** and “**Program.cs**” file in **ASP.NET Core 6.0 and above** as following:

ASP.NET Core 5.0: go to “**Startup.cs**” file and change the code of “**services.AddIdentity**” method which is present in “**ConfigureServices**” method as below:

Old Code:

```

services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
    options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCCoreDbContext>();

```

New Code:

```

services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
    options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCCoreDbContext>().AddDefaultTokenProviders();

```

ASP.NET Core 6.0 and above: go to “**Program.cs**” file and change the code of “**builder.Services.AddIdentity**” method which is present above the statement => “**var app = builder.Build();**” as below:

Old Code:

```

builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
{
    options.Password.RequiredLength = 8;
    options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCCoreDbContext>();

```

New Code:

```

builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
{

```

```

options.Password.RequiredLength = 8;
options.Password.RequireDigit = false;
}).AddEntityFrameworkStores<MVCCoreDbContext>().AddDefaultTokenProviders();

```

After doing all the above changes whenever a new user **registers**, then a **mail** is generated and sent to his registered **mail** with **mail confirmation link** that contains an **Email Confirmation Token** and **User Id. Token** is a unique **"ID"** value that is generated for **email confirmation**, **change email** and **change password**, to identify the user when he clicks on the link.

Step 5: Now with-out **confirming** the **email** a user should not be allowed to **login** into the application and to that, go to **Post Action of Login Method** which is present in **"AccountController"** class and add the below code in top of the if condition i.e., inside **"if (ModelState.IsValid)"** condition:

```

//Code to check whether Email is confirmed or not
var user = await userManager.FindByNameAsync(loginModel.Name);
if (user != null && (await userManager.CheckPasswordAsync(user, loginModel.Password)) &&
    user.EmailConfirmed == false)
{
    ModelState.AddModelError("", "Your email is not confirmed.");
    return View(loginModel);
}

```

Step 6: Now when the user clicks on the **Confirmation Link** in his **mail** it will redirect to **"ConfirmEmail"** action method of **Account Controller** class, so let's define **"ConfirmEmail"** Action Method in **Account Controller** as following:

```

public async Task<IActionResult> ConfirmEmail(string userId, string token)
{
    if(userId != null && token != null)
    {
        var User = await userManager.FindByIdAsync(userId);
        if(User != null)
        {
            var result = await userManager.ConfirmEmailAsync(User, token);
            if(result.Succeeded)
            {
                TempData["Title"] = "Email Confirmation Success.";
                TempData["Message"] = "Email confirmation is completed. You can now login into the application.";
                return View("DisplayMessages");
            }
            else
            {
                StringBuilder Errors = new StringBuilder();
                foreach (var Error in result.Errors)
                {
                    Errors.Append(Error.Description + ". ");
                }
            }
        }
    }
}

```

```

        TempData["Title"] = "Confirmation Email Failure";
        TempData["Message"] = Errors.ToString();
        return View("DisplayMessages");
    }
    else
    {
        TempData["Title"] = "Invalid User Id.";
        TempData["Message"] = "User Id which is present in confirm email link is in-valid.";
        return View("DisplayMessages");
    }
}
else
{
    TempData["Title"] = "Invalid Email Confirmation Link.";
    TempData["Message"] = "Email confirmation link is invalid, either missing the User Id or Confirmation Token.";
    return View("DisplayMessages");
}
}

```

Implementing Password Reset Functionality: Now let's implement **password reset** functionality in our **application** and for that, to do the following:

Step 1: Go to “**Login.cshtml**” and add links for **Register** and **Forgot Password**, just above the “**</form>**” tag as below:

```

<div class="form-group">
    New user? <a asp-action="Register">Click</a> to register? <br />
    Forgot password? - <a asp-action="ForgotPassword">Click</a> to reset.
</div>

```

Step 2: Add a new **Model** class in **Model’s** folder, naming it as “**ChangePasswordModel.cs**” and write the below code in the class:

```

using System.ComponentModel.DataAnnotations;
public class ChangePasswordModel
{
    [Required]
    [Display(Name = "User Name")]
    [RegularExpression("[A-Za-z0-9-._@+]*")]
    public string Name { get; set; }
}

```

Step 3: Go to “**AccountController**” class and write the below Action Methods in the class:

```

public IActionResult ForgotPassword()
{
    return View();
}

```

```

[HttpPost]
public async Task<IActionResult> ForgotPassword(ChangePasswordModel model)
{
    if(ModelState.IsValid)
    {
        var User = await userManager.FindByNameAsync(model.Name);
        if (User != null && await userManager.IsEmailConfirmedAsync(User))
        {
            var token = await userManager.GeneratePasswordResetTokenAsync(User);
            var confirmationUrlLink = Url.Action("ChangePassword", "Account", new { UserId = User.Id, Token = token },
                Request.Scheme);
            SendMail(User, confirmationUrlLink, "Change Password Link");
            TempData["Title"] = "Change Password Link";
            TempData["Message"] = "Change password link has been sent to your mail, click on it and change password.";
            return View("DisplayMessages");
        }
        else
        {
            TempData["Title"] = "Change Password Mail Generation Failed.";
            TempData["Message"] = "Either the Username you have entered is in-valid or your email is not confirmed.";
            return View("DisplayMessages");
        }
    }
    return View(model);
}

```

Now add a View to “ForgotPassword” action method and to do that, choose Razor View in Scaffold Page, click “Add” button, under Template select “Create”, under Model Class select “ChangePasswordModel” and click on “Add” button. Run the View; enter your “Username” to receive a reset password mail. After receiving the mail, user can click on the link to change the password and to perform that we need to define “ResetPassword” Action method.

Step 4: Add another Model class in Model’s folder, naming it as “ResetPasswordModel.cs” and write the below code in it:

```

using System.ComponentModel.DataAnnotations;
public class ResetPasswordModel
{
    [Required]
    public string UserId { get; set; }

    [Required]
    public string Token { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

```

```

[Display(Name = "Confirm Password")]
[DataType(DataType.Password)]
[Compare("Password", ErrorMessage = "Confirm password should match with password.")]
public string ConfirmPassword { get; set; }
}

```

Step 5: Go to “**AccountController**” class and write the below Action Methods in that class:

```

public IActionResult ChangePassword()
{
    return View();
}

[HttpPost]
public async Task<IActionResult> ChangePassword(ResetPasswordModel model)
{
    if (ModelState.IsValid)
    {
        var User = await userManager.FindByIdAsync(model.UserId);
        if (User != null)
        {
            var result = await userManager.ResetPasswordAsync(User, model.Token, model.Password);
            if (result.Succeeded)
            {
                TempData["Title"] = "Reset Password Success";
                TempData["Message"] = "Your password has been reset successfully.";
                return View("DisplayMessages");
            }
            else
            {
                foreach (var Error in result.Errors)
                    ModelState.AddModelError("", Error.Description);
            }
        }
        else
        {
            TempData["Title"] = "Invalid User";
            TempData["Message"] = "No user exists with the given User Id.";
            return View("DisplayMessages");
        }
    }
    return View(model);
}

```

Now add a View to “**ChangePassword**” action method, choose **Razor View** in **Scaffold Page**, click “**Add**”, under **Template** select “**Create**”, under **Model Class** select “**ResetPasswordModel**” and click on “**Add**” button. This view gets launched when we user clicks on the “**Reset Password Email Link**” he has received, and that link contains 2 “**Query String**” values “**UserId & Token**”, and we need to read the values of them into our **page** and to do that modify the code on the top of the view as following:

Old Code:

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

New Code:

```
@{  
    ViewData["Title"] = "ResetPassword";  
    var UserId = @Context.Request.Query["UserId"];  
    var Token = @Context.Request.Query["Token"];  
}
```

Note: In the current view, it will provide “Textbox’s” for entering “User Id & “Token” values, but we are not going to enter those values because those values came to the View as “Query Strings”, when the user clicks on the link that is sent to his email, so delete the 2 “<div>” tags containing controls for entering “User Id” and “Token”. Now we need to bind “User Id & Token” values we have captured, to the Model in the form of “route-values” and to do those change the code of “<form>” tag as following:

Old Code:

```
<form asp-action="ResetPassword">
```

New Code:

```
<form asp-action="ResetPassword" asp-route-userId="@UserId" asp-route-token="@Token">
```

Implementing Open Authentication: open authentication is a process of adding an option of authenticating a User by using external login providers like Google, Facebook, Microsoft, and Twitter.

Implementing Google and Facebook Authentication: to implement Google and Facebook authentication in our application do the following:

Step 1: Re-write the code in “Login.cshtml” as following:

```
@model MVCDHProject.Models.LoginModel  
{@  
    ViewData["Title"] = "Login";  
    var returnUrl = @Context.Request.Query["ReturnUrl"];  
}  
  
<h1>Login With</h1>  
<hr />  
<div class="row">  
    <div class="col-md-6">  
        <h3>NIT Account</h3>  
        <hr />  
        <form asp-action="Login" asp-route-ReturnUrl="@returnUrl">  
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>  
            <div class="form-group">  
                <label asp-for="Name" class="control-label"></label>  
                <input asp-for="Name" class="form-control" />
```

```

        <span asp-validation-for="Name" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="control-label"></label>
        <input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
    <div class="form-group form-check">
        <label class="form-check-label">
            <input class="form-check-input" asp-for="RememberMe" />
            @Html.DisplayNameFor(model => model.RememberMe)
        </label>
    </div>
    <div class="form-group">
        <input type="submit" value="Login" class="btn btn-primary" />
        <input type="reset" value="Reset" class="btn btn-primary" />
    </div>
    <div class="form-group">
        New user? <a asp-action="Register">Click</a> to register?
        <br />
        Forgot password? - <a asp-action="ForgotPassword">Click</a> to reset.
    </div>
</form>
</div>
<div class="col-md-6">
    <h3>Google or Facebook</h3>
    <hr />
    <form asp-action="ExternalLogin" asp-route-returnUrl="@returnUrl">
        <input type="submit" name="Provider" value="Google" class="btn btn-primary" />
        <input type="submit" name="Provider" value="Facebook" class="btn btn-primary" />
    </form>
</div>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Step 2: Register our application under **Google** => to work with **Google**, first we need to **register** our **application** with **Google** and to do that open the site "console.cloud.google.com" and **login** with your **credentials**. Now **click** on **Create Project** => in the window opened enter project name as "**NIT Customer Management**" and click on **Create** button which creates a new project. Now go to "**Navigation Menu**" which is present in **LHS**, on the top and select "**APIS and Services**" => "**Library**" => now in the window opened Search for "**Google+ Api**" which opens a new window, in that click on "**Google+ Api**" and then click on **Enable** button.

Now in the **LHS** top we find "**Google+ API**" and above it we find "**APIs and Services**" click on it which displays the options below, select "**OAuth consent screen**" in **RHS** and click on **Create** button => which will display a screen

and asks for “App Name”, enter the name as “NIT Customer Management”, Enter the Support Email Id: “<enter your email id here>”, scroll down and enter Developer contact information also with your email id and click on “Save and Continue” button.

Now click on “Credentials” in the LHS and click on “Create Credentials” on the top and select “OAuth Client Id” in the provided list, and then choose “Application Type” as “Web Application”, give a name to it, for example: “NIT Client”, and then under “Authorized Java Script Origins”, click on “+Add URI” option and specify the URI for the home page of our site: <http://localhost:port>. Now scroll down to Authorized redirect URLs option click on “+Add URI” option and specify the redirect URI as shown: “YourSiteURI/signin-google” for example: <https://localhost:YourPortNo/signin-google> and click on the Create button which opens a window and in that it will display Client ID and Client Secret, copy and save them because we need to use those values in our Web Application.

Now click on “OAuth consent screen” in LHS and on the right scroll down to Test Users option and click on “ADD USERS” button and add a list of email id’s which you can you in Test Environment.

Note: to get the URI of our site’s home page go to Solution Explorer if our project, right click on the project and select Properties, which opens the project property window and, in that select, “Debug” in LHS and scroll down to the bottom in RHS, select the CheckBox “Enable SSL” which displays the URI, copy, and paste it in Google’s “URI” TextBox. E.g.: <http://localhost:YourPortNo>

Step 3: Registering our application under Facebook => open the site <https://developers.facebook.com> and login with your credentials. Now in the RHS top, click on “MyApps” Menu, which opens a window click on the button “Create App”, which opens another window asking for “Select an app type” => Select “Other” Radio Button and click on Next, select the App Type as “Consumer” and click Next, enter “Display Name” for App, as “NIT Customer Management”, enter “App Contact Email”, click on “Create App” button which will create the app and opens “App Dashboard”.

Now in the “App Dashboard” click on the “Setup” button under “Facebook Login”, which will ask to “Select the platform for this app.”, choose “Web” which will ask for site “URL” enter it as shown: “YourSiteURI/signin-facebook”, for example: <https://localhost:YourPortNo/signin-facebook> and click on “Save”.

Now click on Facebook Login in LHS and under that choose Settings, now on the RHS select the option to configure the client “OAuth Login”, which is already selected “Yes”, so leave the same and under “Valid OAuth Redirect URL’s” enter the URL as “YourSiteURL/signin-facebook” and click on “Save Changes”. Now in the LHS on the top we find “App Settings” option, expand it and under it choose “Basic” which will display “App Id” and “App Secret” in RHS, copy and save because we need to use them in our application.

Now to take your app live we need to provide a privacy policy which should be generated at the following site: <https://www.termsfeed.com>. Login into the site by creating an account, in the window opened select “Private Policy Generator” which opens a window in that choose “Website” and click on Next Step button and fill in the following details:

1. Website URL: <https://localhost:YourPortNo/>
2. Website Name: NIT Customer Management
3. Entity Type: Individual
4. Country: India

5. State: **Telangana**

Click on **Next Step** button and choose the required options (**select all the free options**), click on **Generate a Privacy Policy URL**, then click on **download** button which displays an **URL** copy it. Now come back to **Facebook**, under the “**Basic**” Settings paste the URL in “**Privacy Policy URL**” TextBox and click on **Save Changes** button.

Step 4: come back to your **project** in **Visual Studio** and install the below 2 packages using **NuGet Package Manager**:

Microsoft.AspNetCore.Authentication.Google
Microsoft.AspNetCore.Authentication.Facebook

Step 5: Go to “**Startup.cs**” file in **ASP.NET Core 5.0** and “**Program.cs**” file in **ASP.NET Core 6.0 and above** write the below code to include **Google** and **Facebook** authentications in our application:

ASP.NET Core 5.0: under “**ConfigureServices**” method of **Startup** Class write the below code.

```
services.AddAuthentication()
    .AddGoogle(options =>
{
    options.ClientId = "<Specify Client Id>";
    options.ClientSecret = "<Enter Client Secret>";
})
    .AddFacebook(options =>
{
    options.AppId = "<Specify App Id>";
    options.AppSecret = "<Specify App Secret>";
});
```

ASP.NET Core 6.0 and above: in **Program** Class write the below code just above the statement => “**var app = builder.Build();**”.

```
builder.Services.AddAuthentication()
    .AddGoogle(options =>
{
    options.ClientId = "<Specify Client Id>";
    options.ClientSecret = "<Enter Client Secret>";
})
    .AddFacebook(options =>
{
    options.AppId = "<Specify App Id>";
    options.AppSecret = "<Specify App Secret>";
});
```

Step 6: Now go to “**AccountController**” class and implement the below **Action** methods by importing the namespace “**System.Security.Claims**”:

```
public IActionResult ExternalLogin(string returnUrl, string Provider)
{
```

```

var url = Url.Action("CallBack", "Account", new { ReturnUrl = returnUrl });
var properties = signInManager.ConfigureExternalAuthenticationProperties(Provider, url);
return new ChallengeResult(Provider, properties);
}

public async Task<IActionResult> CallBack(string returnUrl)
{
    if (string.IsNullOrEmpty(returnUrl))
    {
        returnUrl = "~/";
    }
    LoginModel model = new LoginModel();
    var info = await signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ModelState.AddModelError("", "Error loading external login information.");
        return View("Login", model);
    }
    var signInResult = await signInManager.ExternalLoginSignInAsync(info.LoginProvider, info.ProviderKey, false, true);
    if (signInResult.Succeeded)
    {
        return LocalRedirect(returnUrl);
    }
    else
    {
        var email = info.Principal.FindFirstValue(ClaimTypes.Email);
        if (email != null)
        {
            var user = await userManager.FindByEmailAsync(email);
            if (user == null) {
                user = new IdentityUser {
                    UserName = info.Principal.FindFirstValue(ClaimTypes.Email),
                    Email = info.Principal.FindFirstValue(ClaimTypes.Email),
                    PhoneNumber = info.Principal.FindFirstValue(ClaimTypes.MobilePhone),
                };
                var identityResult = await userManager.CreateAsync(user);
            }
            await userManager.AddLoginAsync(user, info);
            await signInManager.SignInAsync(user, false);
            return LocalRedirect(returnUrl);
        }
        TempData["Title"] = "Error";
        TempData["Message"] = "Email claim not received from third party provided.";
        return RedirectToAction("DisplayMessages");
    }
}

```

Hosting MVC Core Application on Microsoft Azure

Open the site “[portal.azure.com](#)” and create an account to login into the site. Click on the **Subscriptions** button on the **home** screen which opens another window and in that click on “**+ Add**” button which opens a new window and, in that select, “**Azure for Students**” and click on “**Select Offer**” button which opens a new window, click on “**Activate**” button and provide all the required details.

Now open our project “**MVCDHProject**” in **Visual Studio**, then go to **Solution Explorer**, right click on the project and select the option “**Publish**” this opens the **Publish Window** and in that under “**Target**” choose “**Azure**” and click next, now under the “**Specific Target**” choose “**Azure App Service (Windows)**” and click next, now under the “**App Service**” we need to create an “**App Service Instance**” and to do that click on “**+ (Create an Azure App Service)**” button which opens a window and in that provide the following details:

Name: nitcore

Subscription: “Choose your subscription plan here”

Resource Group: Click on New button beside and enter a name to it, for example “**NITRG**”.

Hosting Plan: Click on New button besides, which opens a window and in that enter the following details:

Hosting Plan: **NITHP**

Location: “Choose any location or leave the default”

Size: “Choose the size or leave the default”

=> Click on the **Ok** button

=> Click on the **Create** button

=> Click on the **Finish** button

Now in the publish window, it will recognize all the dependencies, and our application right now has a “**Database Dependency**”, so it will display that in the bottom of the window under “**Service Dependencies**” option, click on the “**+ (Add new service dependency)**” button and in the window opened, select “**Azure SQL Database**” option and click “**Next**” button, now in the new window opened click on the “**+ (Create a SQL Database)**” button which opens a new window, provide the following details in it:

Database name: Enter a name to the Database or leave the existing name

Subscription: Choose your subscription plan here

Resource Group: NITRG

Database Server: Click on the New button beside and enter the following details in the window opened:

Database server name: nitdbserver

Location: Choose any location or leave the default

Administrator username: NitAdmin

Administrator password: NIT2024pwd

=> Click on the **Ok** button

=> Click on the **Create** button

=> Click on the **Next** button which will ask for **Connection String** details, so enter the following in it:

Database connection string name: ConStr

Database connection username: NitAdmin

Database connection password: NIT2024pwd

=> Click on the **Finish** button which will finish the configuration, click on the **Close** button, and close.

Now in the **Publish** window we find “**More actions**” dropdown select “**Edit**” in it which opens a new window and in that on the **LHS** choose “**Settings**” and do the following:

- Expand the “**Databases**” option and check the CheckBox “**Use this connection string at runtime**”.
 - Expand the “**Entity Framework Migrations**” option and check the CheckBox “**Apply this migration on publish**”.
- => Click on the **Save** button which closes the window.

=> In the **Publish** window click on the **Publish** button which will upload the site to azure server, which we can now access by using the URL: <https://nitcore.azurewebsites.net>

Entity Framework Core DB First

This was available in **Entity Framework Core** also but with a very limited support. To work with **DB First**, create a new “**ASP.NET Core Web App (Model-View-Controller)**” Project, naming it as “**MVCCoreDBF**”, choose “**.NET 8.0 (Long Term Support)**”, Select the Checkbox “**Configure for HTTPS**”, Select the Checkbox “**Do not use top-level statements**” and click on “**Create**” button.

Note: in **EF Core** we don’t have any **designer support** for creating **DB First** project just like what we have in **EF 6**. So here we need to use **Scaffolding** commands to generate **Model** and **Context** classes.

To generate Model and Context classes use the below Scaffold command:

`Scaffold-DbContext [ConnectionString] [Provider] [-OutputDir <Name>] [-Tables <Name>]`

Step 1: To use **Scaffold Commands** and **Entity Framework Core** for **SQL Server** first we need to install the below packages using **NuGet Package Manager**:

`Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.SqlServer`

Step 2: Open **PMC (Package Manager Console)** and write the below code at the “**PM**” command prompt.

```
PM> Scaffold-DbContext "Data Source=Server;User Id=Sa;Password=123;Database=MVCDB;  
TrustServerCertificate=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Student
```

Note: in the **Scaffold Command** if we don’t use “[**-Tables**] <Table_Name>” attribute it will generate **Model** classes for all the **Tables** in the **Database**.

The above statement will generate the **Model** class with the name “**Student**” and also generates the **Context** class with the name “**MVCDBContext**” as below:

```
public partial class Student  
{  
    public int Sid { get; set; }  
    public string Name { get; set; }  
    public int? Class { get; set; }  
    public decimal? Fees { get; set; }  
    public string Photo { get; set; }  
    public bool? Status { get; set; }  
}
```

```

public partial class MVCDBContext : DbContext
{
    public MVCDBContext() {
    }
    public MVCDBContext(DbContextOptions<MVCDBContext> options) : base(options) {
    }
    public virtual DbSet<Student> Students { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            #warning To protect potentially sensitive information in your connection string, you should move it out of
            source code. You can avoid scaffolding the connection string by using the Name= syntax to read it from
            configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For more guidance on storing
            connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
            optionsBuilder.UseSqlServer("Data Source=Server;Database=MVCDB;User Id=Sa;Password=123");
        }
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasAnnotation("Relational:Collation", "SQL_Latin1_General_CI_AS");
        modelBuilder.Entity<Student>(entity =>
        {
            entity.HasKey(e => e.Sid).HasName("PK__Student__CA1E5D78E2EF83E9");
            entity.ToTable("Student");
            entity.Property(e => e.Sid).ValueGeneratedNever();
            entity.Property(e => e.Fees).HasColumnType("money");
            entity.Property(e => e.Name).HasMaxLength(50).IsUnicode(false);
            entity.Property(e => e.Photo).HasMaxLength(100).IsUnicode(false);
            entity.Property(e => e.Status).IsRequired().HasDefaultValueSql("(1)");
        });
        OnModelCreatingPartial(modelBuilder);
    }
    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}

```

Step 3: To define the **Controller** class, **Action** methods and **Views** also let us use **Scaffolding** and to do that, right click on the **Controllers** folder, Select **Add => Controller** and in the window opened select “**MVC Controller with views, using Entity Framework**” and click **Add** button, which will open a new window and in that select the “**Model cls:**” as “**Student**” and “**Data context class:**” as “**MVCDBContext (MVCCoreDBF.Models)**” and click on the “**Add**” button which will generate the **Controller** class with all the required action methods and corresponding **Views**, and the name of the **Controller** will be “**StudentsController**”.

Step 4: Now go to “**Startup**” class in case of **ASP.NET Core 5.0** or “**Program**” class in case of **ASP.NET Core 6.0** and register the “**DbContext**” class by calling “**AddDbContext**” method and to do this first we need to import the namespace “**MVCCoreDBF.Models**” and write the code.

ASP.NET Core 5.0: write the below statement under “ConfigureServices” method of “Startup” class:
services.AddDbContext<MVCDBContext>();

ASP.NET Core 6.0 and above: write the below statement under “Program” class just above the statement => “var app = builder.Build();”:
builder.Services.AddDbContext<MVCDBContext>();

Now run the **project** and watch the output of all the **Views**, but the problem is with the **image** i.e., it will not be displayed and in “Add” and “Edit” Views we will not find the option for selecting the **image**, so we need to now make all the necessary changes for including the **Image** in **Display**, **New**, **Edit** and **Delete** Views.

Step 5: Create a folder with the name “**images**” under “**wwwroot**” folder and copy all the image files of **Students** into that folder.

Step 6: Now make the changes in all the required **Views** and their corresponding **Action** methods as following:

Index View: go to “**Index.cshtml**” and replace the code of **Photo** property with new code as following:

Old Code: @Html.DisplayFor(modelItem => item.Photo)
New Code:

Details View: go to “**Details.cshtml**” and replace the code of **Photo** property with new code as following:

Old Code: @Html.DisplayFor(model => model.Photo)
New Code:

Create View: go to “**Create.cshtml**” and replace the code of **Photo** property with new code as following:

Old Code: <input asp-for="Photo" class="form-control" />
New Code: <input type="file" name="selectedFile" />

Edit View: go to “**Edit.cshtml**” and replace the code of **Photo** property with new code as following:

Old Code: <input asp-for="Photo" class="form-control" />
New Code:
 <input type="file" name="selectedFile" />

Note: in “**Create.cshtml**” and “**Edit.cshtml**”, add “**enctype**” attribute to “**<form>**” tag which should look as below:

Create.cshtml: <form asp-action="Create" enctype="multipart/form-data">
Edit.cshtml: <form asp-action="Edit" enctype="multipart/form-data">

Delete View: go to “**Delete.cshtml**” and replace the code of **Photo** property with new code as following:

Old Code: @Html.DisplayFor(model => model.Photo)
New Code:

Now go to “**StudentsController**” class, import “**Microsoft.AspNetCore.Mvc.ModelBinding**” namespaces and make the below changes:

Re-write the constructor in the class by declaring a new field as following:

```
private readonly IWebHostEnvironment _environment;
```

```

public StudentsController(MVCDBContext context, IWebHostEnvironment environment) {
    _context = context;
    _environment = environment;
}

```

Re-write the Post Action method of Create as following:

```

public async Task<IActionResult> Create([Bind("Sid,Name,Class,Fees,Photo,Status")] Student student,
                                         IFormFile selectedFile)
{
    if (ModelState.IsValid)
    {
        if (selectedFile != null)
        {
            string FolderPath = _environment.WebRootPath + "\\images";
            if (!Directory.Exists(FolderPath))
            {
                Directory.CreateDirectory(FolderPath);
            }
            string FilePath = FolderPath + "\\" + selectedFile.FileName;
            FileStream fs = new FileStream(FilePath, FileMode.Create);
            selectedFile.CopyTo(fs);
            student.Photo = selectedFile.FileName;
        }
        student.Status = true;
        _context.Add(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(student);
}

```

Re-write the Get Action method of Edit as following:

```

public async Task<IActionResult> Edit(int? id)
{
    if (id == null || _context.Students == null) {
        return NotFound();
    }
    var student = await _context.Students.FindAsync(id);
    if (student == null) {
        return NotFound();
    }
    if (student.Photo != null) {
        TempData["Photo"] = student.Photo;
    }
    return View(student);
}

```

Re-write the Post Action method of Edit as following:

```
public async Task<IActionResult> Edit(int id, [Bind("Sid,Name,Class,Fees,Photo,Status")] Student student, IFormFile selectedFile)
{
    if (id != student.Sid)
    {
        return NotFound();
    }
    if (ModelState.IsValid || (ModelState.ErrorCount == 1 && ModelState["selectedFile"].ValidationState == ModelValidationState.Invalid))
    {
        try
        {
            if (selectedFile != null)
            {
                string FolderPath = Path.Combine(_environment.WebRootPath, "images");
                if (!Directory.Exists(FolderPath))
                {
                    Directory.CreateDirectory(FolderPath);
                }
                string ImagePath = Path.Combine(FolderPath, selectedFile.FileName);
                FileStream fs = new FileStream(ImagePath, FileMode.Create);
                selectedFile.CopyTo(fs);
                student.Photo = selectedFile.FileName;
            }
            else if (TempData["Photo"] != null)
            {
                student.Photo = TempData["Photo"].ToString();
            }
            _context.Update(student);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!StudentExists(student.Sid))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(student);
}
```

Web API Core

Create a new project of type “ASP.NET Core Web API”, name the project as “CoreWeb ApiService”, choose Target Framework as “.NET 8.0 (Long-Term support)” and check these Checkbox’s “Configure for HTTPS”, “Use controllers (uncheck to use minimal APIs)”, “Enable Open API Support” and “Do not use top-level statements”.

Let's create an API Service that uses XML as Data Source and to do that follow the below process:

Step 1: add an XML File in the project naming it as “Customer.xml” and write the below code in it:

```
<Customers>
  <Customer>
    <Custid>101</Custid>
    <Name>Scott</Name>
    <Balance>25000</Balance>
    <City>Hyderabad</City>
    <Status>True</Status>
  </Customer>
  <Customer>
    <Custid>102</Custid>
    <Name>Smith</Name>
    <Balance>35000</Balance>
    <City>Kolkata</City>
    <Status>True</Status>
  </Customer>
  <Customer>
    <Custid>103</Custid>
    <Name>David</Name>
    <Balance>45000</Balance>
    <City>Bengaluru</City>
    <Status>True</Status>
  </Customer>
  <Customer>
    <Custid>104</Custid>
    <Name>Sofia</Name>
    <Balance>55000</Balance>
    <City>Mumbai</City>
    <Status>True</Status>
  </Customer>
</Customers>
```

Step 2: Add a new folder under the project naming it as “Models” and then under the folder add a class naming it as “Customer” and write the below code in it:

```
public class Customer
{
  public int Custid { get; set; }
  public string Name { get; set; }
  public decimal? Balance { get; set; }
```

```

public string City { get; set; }
public bool Status { get; set; }
}

```

Step 3: Add an **interface** in the **Models** folder naming it as “**ICustomerDAL**” so that **DAL Class** implementation can use **Dependency Injection Pattern**, and write the below code in the **interface**:

```

public interface ICustomerDAL
{
    List<Customer> Customers_Select();
    Customer Customer_Select(int Custid);
    void Customer_Insert(Customer customer);
    void Customer_Update(Customer customer);
    void Customer_Delete(int Custid);
}

```

Step 4: Add a **class** in the **Models** folder naming it as “**CustomerXmlDAL**”, and right now we are implementing the **DAL** class logic to work with **XML Data Source**, and to do that write the below code in the class:

```

using System.Data;
public class CustomerXmlDAL : ICustomerDAL
{
    DataSet ds;
    public CustomerXmlDAL()
    {
        ds = new DataSet();
        ds.ReadXml("Customer.xml");
        ds.Tables[0].PrimaryKey = new DataColumn[] { ds.Tables[0].Columns["Custid"] };
    }
    public List<Customer> Customers_Select()
    {
        List<Customer> Customers = new List<Customer>();
        foreach (DataRow dr in ds.Tables[0].Rows)
        {
            Customer obj = new Customer
            {
                Custid = Convert.ToInt32(dr["Custid"]),
                Name = (string)dr["Name"],
                Balance = Convert.ToDecimal(dr["Balance"]),
                City = (string)dr["City"],
                Status = Convert.ToBoolean(dr["Status"])
            };
            Customers.Add(obj);
        }
        return Customers;
    }
}

```

```

public Customer Customer_Select(int Custid)
{
    DataRow dr = ds.Tables[0].Rows.Find(Custid);
    if (dr != null)
    {
        Customer obj = new Customer
        {
            Custid = Convert.ToInt32(dr["Custid"]),
            Name = (string)dr["Name"],
            Balance = Convert.ToDecimal(dr["Balance"]),
            City = (string)dr["City"],
            Status = Convert.ToBoolean(dr["Status"])
        };
        return obj;
    }
    return null;
}
public void Customer_Insert(Customer customer)
{
    DataRow dr = ds.Tables[0].NewRow();
    dr[0] = customer.Custid;
    dr[1] = customer.Name;
    dr[2] = customer.Balance;
    dr[3] = customer.City;
    dr[4] = customer.Status;
    ds.Tables[0].Rows.Add(dr);
    ds.WriteXml("Customer.xml");
}
public void Customer_Update(Customer customer)
{
    DataRow dr = ds.Tables[0].Rows.Find(customer.Custid);
    int Index = ds.Tables[0].Rows.IndexOf(dr);
    ds.Tables[0].Rows[Index]["Name"] = customer.Name;
    ds.Tables[0].Rows[Index]["Balance"] = customer.Balance;
    ds.Tables[0].Rows[Index]["City"] = customer.City;
    ds.Tables[0].Rows[Index]["Status"] = customer.Status;
    ds.WriteXml("Customer.xml");
}
public void Customer_Delete(int Custid)
{
    DataRow dr = ds.Tables[0].Rows.Find(Custid);
    int Index = ds.Tables[0].Rows.IndexOf(dr);
    ds.Tables[0].Rows[Index].Delete();
    ds.WriteXml("Customer.xml");
}
}

```

Step 5: Go to “`Startup.cs`” file in case ASP.Net Core 5.0 and `Program.cs` file in case of ASP.NET Core 6.0 and write the below statement by importing “`CoreWebApiService.Models`” namespace:

ASP.NET Core 5.0 => go to `ConfigureServices` method in `Startup` class and write the below statement over there:

```
builder.Services.AddScoped(typeof(ICustomerDAL), typeof(CustomerXmlDAL));
```

ASP.NET Core 6.0 and above => go to `Main` method in `Program` class and write the below statement just above the statement “`var app = builder.Build();`”:

```
builder.Services.AddScoped(typeof(ICustomerDAL), typeof(CustomerXmlDAL));
```

Step 6: Add an “`ApiController`” in the `Controllers` folder and to do that right click on the “`Controllers`” folder, select `Add => Controller`, in the window opened select `API` in `LHS` and then on the `RHS` select “`API Controller - Empty`”, click `Add` button, name the class as “`CustomerController`” and write the below code in the `class`:

```
using System.Net;
using System.Net.Http;
using CoreWebApiService.Models;

[Route("api/[controller]")]
[ApiController]
public class CustomerController : ControllerBase
{
    private readonly ICustomerDAL dal;
    public CustomerController(ICustomerDAL dal)
    {
        this.dal = dal;
    }
    [HttpGet]
    public List<Customer> GetCustomers()
    {
        return dal.Customers_Select();
    }
    [HttpGet("{Custid}")]
    public Customer GetCustomer(int Custid)
    {
        return dal.Customer_Select(Custid);
    }
    [HttpPost]
    public HttpResponseMessage Post(Customer c)
    {
        dal.Customer_Insert(c);
        return new HttpResponseMessage(HttpStatusCode.Created);
    }
    [HttpPut]
    public HttpResponseMessage Put(Customer c)
    {
```

```

Customer customer = dal.Customer_Select(c.Custid);
if (customer != null)
{
    dal.Customer_Update(c);
    return new HttpResponseMessage(HttpStatusCode.OK);
}
else
{
    return new HttpResponseMessage(HttpStatusCode.NotFound);
}
}

[HttpDelete("{Custid}")]
public HttpResponseMessage Delete(int Custid)
{
    Customer customer = dal.Customer_Select(Custid);
    if (customer != null)
    {
        dal.Customer_Delete(Custid);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }
    else
    {
        return new HttpResponseMessage(HttpStatusCode.NotFound);
    }
}
}

```

To test **Web API Core**, we are provided with a built-in tool known as “**Swagger**”, so we don’t require using any web debugging tools like **Postman** or **Fiddler** etc. Run the “**APIController**” we have defined and test all the functionalities by using **Swagger**.

Consuming our Core API Service in an MVC Controller using JQuery-Ajax with-in the Same Project: add a new Controller in the current project and it should be an “**MVC Controller**”, naming it as “**HomeController**” and add a **View** to the default **Index** action method and write the below code in it:

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Web Api Core</title>
<script src="https://code.jquery.com/jquery-3.7.0.min.js"
       integrity="sha256-2Pmvv0kuTBOenSvLm6bvfBSSHrUJ+3A7x6P5Ebd07/g="
       crossorigin="anonymous"></script>
<script>
$(document).ready(function () {
    GetCustomers();
});

```

```

function GetCustomers() {
    $.ajax({
        url: 'http://localhost/CoreWeb ApiService/api/Customer',
        type: 'GET',
        dataType: 'json',
        success: function (Customers) {
            $("#tblCustomers tbody").empty();
            $("#tblCustomers").append("<tr><td><input id='custid' type='text' style='width: 50px' /></td><td><input id='name' type='text' /></td><td><input id='balance' type='text' /></td><td><input id='city' type='text' /></td><td align='center'><input id='status' type='checkbox' /></td><td align='center'><input type='button' id='insert' value='Insert' onclick='AddCustomer()' /></td></tr>");
            $.each(Customers, function (index, Customer) {
                $("#tblCustomers").append(BuildCustomerRow(Customer));
            });
        },
        error: function (request) {
            HandleException(request);
        }
    });
}

function BuildCustomerRow(Customer) {
    var NewRow = "<tr>" + "<td align='center'>" + Customer.custid + "</td>" +
        "<td><input type='text' class='input-name' value='" + Customer.name + "' /></td>" +
        "<td><input type='text' class='input-balance' value='" + Customer.balance + "' /></td>" +
        "<td><input type='text' class='input-city' value='" + Customer.city + "' /></td>";
    if (Customer.status == true)
        NewRow += "<td align='center'><input class='input-status' type = 'checkbox' checked /></td>";
    else
        NewRow += "<td align='center'><input class='input-status' type = 'checkbox' /></td>";

    NewRow += "<td><button type='button' onclick='UpdateCustomer(this);' data-custid='" +
        Customer.custid + "'>Update</button>";
    NewRow += "<button type='button' onclick='DeleteCustomer(this);' data-custid='" +
        Customer.custid + "'>Delete</button>";
    NewRow += "</td></tr>";
    return NewRow;
}

function HandleException(request) {
    var msg = "";
    msg += "Code: " + request.status + "\n";
    msg += "Text: " + request.statusText + "\n";
    if (request.responseJSON != null) {
        msg += "Message" + request.responseJSON.Message + "\n";
    }
    alert(msg);
}

```

```

var Customer = {
    custid: 0,
    name: "",
    balance: 0,
    city: "",
    status: false
}
function AddCustomer() {
    var obj = Customer;
    obj.custid = $("#custid").val();
    obj.name = $("#name").val();
    obj.balance = $("#balance").val();
    obj.city = $("#city").val();
    obj.status = $("#status").is(":checked");

    var options = {};
    options.url = "http://localhost/CoreWeb ApiService/api/Customer";
    options.type = "POST";
    options.contentType = "application/json";
    options.dataType = "json";
    options.data = JSON.stringify(obj);
    options.success = function () {
        GetCustomers();
        $("#divMsgs").html("Insert operations is successful.");
    }
    options.error = function () {
        $("#divMsgs").html("Error while performing insert operation!");
    }
    $.ajax(options);
}

function UpdateCustomer(button) {
    var obj = Customer;
    obj.custid = $(button).data("custid");
    obj.name = $(".input-name", $(button).parent().parent()).val();
    obj.balance = $(".input-balance", $(button).parent().parent()).val();
    obj.city = $(".input-city", $(button).parent().parent()).val();
    obj.status = $(".input-status", $(button).parent().parent()).is(":checked");

    var options = {};
    options.url = "http://localhost/CoreWeb ApiService/api/customer";
    options.type = "PUT";
    options.contentType = "application/json";
    options.dataType = "json";
    options.data = JSON.stringify(obj);
    options.success = function () {
        GetCustomers();
    }
}

```

```

        $("#divMsgs").html("Update operations is successful.");
    }
    options.error = function () {
        $("#divMsgs").html("Error while performing update operation!");
    }
    $.ajax(options);
}
function DeleteCustomer(button) {
$.ajax({
    url: 'http://localhost/CoreWeb ApiService/api/customer/' + $(button).data("custid"),
    type: 'DELETE',
    dataType: 'json',
    success: function () {
        GetCustomers();
        $("#divMsgs").html("Delete operations is successful.");
    },
    error: function () {
        $("#divMsgs").html("Error while performing delete operation!");
    }
});
}

```

</script>

</head>

<body style="background-color:cornflowerblue">

<form>

<table align="center" id="tblCustomers" border="1">

<thead>

<tr>

<th>Custid</th>

<th>Name</th>

<th>Balance</th>

<th>City</th>

<th>Status</th>

<th>Actions</th>

</tr>

</thead>

<tbody></tbody>

</table>

<div id="divMsgs" style="color:red"></div>

</form>

</body>

</html>

Right now, our project is a “**WebApiProject**” so we can run only “**ApiController**” whereas if we want to run our “**MVCController**” do the following:

Step 1: go to “Startup.cs” file in **ASP.NET Core 5.0** or “Program.cs” file in **ASP.NET Core 6.0** and change the method call “AddControllers” as “AddControllersWithViews” as below:

ASP.NET Core 5.0: go to **ConfigureServices** method in **Startup** class and do the below change:

<u>Old Code</u>	=> services.AddControllers();
<u>New Code</u>	=> services.AddControllersWithViews();

ASP.NET Core 6.0 and above: go to **Program** class and do the below change:

<u>Old Code</u>	=> builder.Services.AddControllers();
<u>New Code</u>	=> builder.Services.AddControllersWithViews();

Step 2: also change the “**EndPoint**” configuration code as below:

ASP.NET Core 5.0:

Old Code:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

New Code:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}")
    );
    endpoints.MapControllers();
});
```

ASP.NET Core 6.0 and above: add this code just above the method call “**app.MapControllers();**”:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Step 3: Go to “**Web.config**” file in the project and there we find a node “**<system.webServer>**” and under that we find “**<handlers>**” node, write the below statement inside of that **<handlers>** node:

```
<remove name="WebDAV" />
```

➤ Now under “**<system.webServer>**” node if you find “**<modules>**” node, add the below statement inside of it:

```
<remove name="WebDAVModule" />
```

➤ If you don’t find “**<modules>**” node, then write the below statements:

```
<modules><remove name="WebDAVModule" /></modules>
```

Step 4: host the “**API Service on IIS**” and then run the project using “**IIS Web Server**”.

Step 5: run the “**HomeController’s - Index**” action method and test all the functionalities.

Consuming our API Service in an MVC Controller using JQuery Ajax from Another Project: If we want to consume an “API Service” from other projects first we need to enable “CORS” in our “API Service Project” and to do that make the below changes:

ASP.NET Core 5.0 => go to “ConfigureServices” method of “Startup.cs” file and add “Cors Service” as below:
services.AddCors();

ASP.NET Core 6.0 and above => go to “Program.cs” file and add “Cors Service” just above the statement “var app = builder.Build();”:

```
builder.Services.AddCors();
```

ASP.NET Core 5.0 => go to “Configure” method of “Startup.cs” file and add “Cors Middleware” as below (write this statement after “app.UseRouting();”) statement:

```
app.UseCors(options =>
{
    options.AllowAnyOrigin();
    options.AllowAnyMethod();
    options.AllowAnyHeader();
});
```

ASP.NET Core 6.0 and above => go to “Program.cs” file and add “Cors Middleware” just below the statement “var app = builder.Build();”:

```
app.UseCors(options =>
{
    options.AllowAnyOrigin();
    options.AllowAnyMethod();
    options.AllowAnyHeader();
});
```

Now create a new “ASP.NET Core Web App (Model-View-Controller)” project naming it as “CoreWebApiConsumer1”. Add a **Controller** in the project naming it as “TestApiController”, add a View to the default “Index” action method, copy the whole code of “Index View” present in “HomeController” of our previous project i.e., “CoreWeb ApiService”, paste it in “Index View” of the current project and run it.

Consuming our API Service in an MVC Controller using C# Code from Another Project: In the above case we have seen how to consume an API Service in an MVC Project using JQuery AJAX, whereas if we want to consume it from Action Methods of an MVC Controller using C#, do the following.

Step 1: Create a “ASP.NET Core Web App (Model-View-Controller)” project naming it as “CoreWebApiConsumer2”, add a new class in the **Models** folder naming it as “Customer” and write the below code in the class:

```
public class Customer
{
    public int Custid { get; set; }
```

```

public string Name { get; set; }
public decimal? Balance { get; set; }
public string City { get; set; }
public bool Status { get; set; }
}

```

Step 2: Install “Microsoft.AspNet.WebApi.Client” library using [NuGet Package Manager](#).

Step 3: Add a **Controller** in the project naming it as “**TestApiController**”, delete all the existing code in the class and write the below code in the class:

```

using Newtonsoft.Json;
using CoreWebApiConsumer2.Models;
public class TestApiController : Controller
{
    HttpClient client = new HttpClient();
    String serviceUri = "http://localhost/CoreWebApiService/api/";
    public async Task<IActionResult> DisplayCustomers()
    {
        List<Customer> customers = new List<Customer>();
        client.BaseAddress = new Uri(serviceUri);
        HttpResponseMessage response = await client.GetAsync("Customer");
        if (response.IsSuccessStatusCode)
        {
            string result = response.Content.ReadAsStringAsync().Result;
            customers = JsonConvert.DeserializeObject<List<Customer>>(result);
        }
        return View(customers);
    }
    public async Task<IActionResult> DisplayCustomer(int Custid)
    {
        Customer customer = new Customer();
        client.BaseAddress = new Uri(serviceUri);
        HttpResponseMessage response = await client.GetAsync("Customer/" + Custid);
        if (response.IsSuccessStatusCode)
        {
            string result = response.Content.ReadAsStringAsync().Result;
            customer = JsonConvert.DeserializeObject<Customer>(result);
        }
        return View(customer);
    }
    public IActionResult AddCustomer()
    {
        return View();
    }
}

```

```

[HttpPost]
public async Task<IActionResult> AddCustomer(Customer customer)
{
    client.BaseAddress = new Uri(serviceUri);
    HttpResponseMessage response = await client.PostAsJsonAsync("Customer", customer);
    if (response.IsSuccessStatusCode)
        return RedirectToAction("DisplayCustomers");
    else
        return View();
}

public async Task<IActionResult> EditCustomer(int Custid)
{
    Customer customer = new Customer();
    client.BaseAddress = new Uri(serviceUri);
    HttpResponseMessage response = await client.GetAsync("Customer/" + Custid);
    if (response.IsSuccessStatusCode)
    {
        string result = response.Content.ReadAsStringAsync().Result;
        customer = JsonConvert.DeserializeObject<Customer>(result);
    }
    return View(customer);
}

public async Task<IActionResult> UpdateCustomer(Customer customer)
{
    client.BaseAddress = new Uri(serviceUri);
    HttpResponseMessage response = await client.PutAsJsonAsync("Customer", customer);
    if (response.IsSuccessStatusCode)
        return RedirectToAction("DisplayCustomers");
    else
        return View("EditCustomer");
}

public async Task<IActionResult> DeleteCustomer(int Custid)
{
    client.BaseAddress = new Uri(serviceUri);
    HttpResponseMessage response = await client.DeleteAsync("Customer/" + Custid);
    if (!response.IsSuccessStatusCode)
        ModelState.AddModelError("", "Delete action resulted in an error");
    return RedirectToAction("DisplayCustomers");
}

```

Step 4: Add Views to “DisplayCustomers”, “DisplayCustomer”, “AddCustomers” and “EditCustomer” actions methods and write the below code over there:

DisplayCustomers.CShtml:

```
@model IEnumerable<Customer>
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" align="center" class="table-condensed">
<tr>
<th>@Html.DisplayNameFor(C => C.Custid)</th>
<th>@Html.DisplayNameFor(C => C.Name)</th>
<th>@Html.DisplayNameFor(C => C.Balance)</th>
<th>@Html.DisplayNameFor(C => C.City)</th>
<th>@Html.DisplayNameFor(C => C.Status)</th>
<th>Actions</th>
</tr>
@foreach (Customer customer in Model)
{
<tr>
<td align="center">@Html.DisplayFor(C => customer.Custid)</td>
<td>@Html.DisplayFor(C => customer.Name)</td>
<td>@Html.DisplayFor(C => customer.Balance)</td>
<td>@Html.DisplayFor(C => customer.City)</td>
<td align="center">@Html.DisplayFor(C => customer.Status)</td>
<td>
<a asp-action="DisplayCustomer" asp-route-Custid="@customer.Custid">View</a> &nbsp;
<a asp-action="EditCustomer" asp-route-Custid="@customer.Custid">Edit</a> &nbsp;
<a asp-action="DeleteCustomer" asp-route-Custid="@customer.Custid"
    onclick="return confirm('Are you sure of deleting the record?')">Delete</a>
</td>
</tr>
}
<tr>
<td colspan="6" align="center"><a asp-action="AddCustomer">Add New Customer</a></td>
</tr>
</table>
```

DisplayCustomer.cshtml:

```
@model Customer
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" align="center">
<tr><td>Custid:</td><td>@Model.Custid</td></tr>
<tr><td>Name:</td><td>@Model.Name</td></tr>
<tr><td>Balance:</td><td>@Model.Balance</td></tr>
<tr><td>City:</td><td>@Model.City</td></tr>
<tr><td>Status:</td><td>@Model.Status</td></tr>
</table>
<div style="text-align:center">
<a asp-action="DisplayCustomers" align="center">Back to Customer Details</a>
</div>
```

AddCustomer.cshtml:

```
@model Customer
<form asp-controller="TestApi" asp-action="AddCustomer" method="post">
    <div><label asp-for="Custid"></label><br /><input asp-for="Custid" /></div>
    <div><label asp-for="Name"></label><br /><input asp-for="Name" /></div>
    <div><label asp-for="Balance"></label><br /><input asp-for="Balance" /></div>
    <div><label asp-for="City"></label><br /><input asp-for="City" /></div>
    <div><label asp-for="Status"></label><br /><input asp-for="Status" /></div>
    <div>
        <input type="submit" value="Save" />
        <input type="reset" value="Reset" />
    </div>
</form>
<div>
    <a asp-action="DisplayCustomers" align="center">Back to Customer Details</a>
</div>
```

EditCustomer.cshtml:

```
@model Customer
<form asp-controller="TestApi" asp-action="UpdateCustomer" method="post">
    <div><label asp-for="Custid"></label><br /><input asp-for="Custid" readonly /></div>
    <div><label asp-for="Name"></label><br /><input asp-for="Name" /></div>
    <div><label asp-for="Balance"></label><br /><input asp-for="Balance" /></div>
    <div><label asp-for="City"></label><br /><input asp-for="City" /></div>
    <div><label asp-for="Status"></label><br /><input asp-for="Status" /></div>
    <div>
        <input type="submit" value="Update" />
        <input type="reset" value="Reset" />
    </div>
</form>
<div>
    <a asp-action="DisplayCustomers" align="center">Back to Customer Details</a>
</div>
```

Integrating Angular into MVC Core Projects

Step 1: Visit the site <https://nodejs.org/en/> and install the latest version of node, which will also install the tool “npm (Node Package Manager)” on our machines. After installation is completed, open “Windows Command Prompt”, and use the below commands to check the versions of “node” and “npm”.

Checking the version of node: node -v

Checking the version of npm: npm -v

Step 2: Install “Type Script” and “Angular CLI” by using the below commands on Windows Command Prompt:

Installing Type Script: npm install -g typescript

Installing Angular CLI: npm install -g @angular/cli

Step 3: Create a new “ASP.NET Core Web App (Model-View-Controller)” project naming it as “AngularInMVCCore”. Add a new folder under the project naming it as “Angular”, right click on the new folder and select “Open Folder in File Explorer”, this opens a window, in that window in the Path TextBox, enter “cmd” and hit enter which will open Windows Command Prompt pointing to the folder location and, in this folder, we are going to create an Angular Project with the help of “Angular CLI” using the command “ng new <Project Name>”, for example:

```
<drive>:\<Personal Folder>\AngularInMVCCore\AngularInMVCCore\Angular> ng new FirstProject
```

This command when executed will ask for adding Angular Routing, choose “No”, and then it will ask which Style Sheet Format you would like to use, choose “CSS”. This will install Angular in the “FirstProject” folder which is created under “Angular” folder.

Now change to the new folder by using “cd FirstProject” which will change us to:

```
<drive>:\<Personal Folder>\AngularInMVCCore\AngularInMVCCore\Angular\FirstProject>
```

If you want to run the Angular Project and test it use the below command and hit enter:

```
<drive>:\<Personal Folder>\AngularInMVCCore\AngularInMVCCore\Angular\FirstProject> ng s -o
```

This will open browser and displays the output, and right now this output is coming from “app.component.html” file which is present under “FirstProject” we have created and to check that go to “Visual Studio” and under “Angular” Folder we find the project Folder i.e., “FirstProject” and under that we will find “src” folder and under that we find “app” folder which will contain the file “app.component.html”, open it, delete the whole content in it and write the below code over there:

```
<h1>Hello World</h1>
```

Now come back to Command Prompt and run the project again by using “ng s -o” command, which will display the modified output i.e., “Hello World”.

Step 4: To integrate Angular with MVC we need to do some changes in “angular.json” file of “FirstProject” which is the configuration file of the project. By default when we build an Angular Project it will generate output files in “dist/first-project” folder which is created under the “FirstProject” folder but if we want to integrate Angular with MVC they should be stored under “wwwroot” folder, so we need to create a folder under “wwwroot” with the name “AngularScripts” and then storing the generated build files into that folder and to do that change the location of Output Path Directory in “angular.json” file as following:

Old Value => "outputPath": "dist/FirstProject"

New Value => "outputPath": "../wwwroot/AngularScripts"

By default every **output file** that is generated by **Angular** will be having an “**hash key**” suffixed to it and that “**hash key**” value will be changing whenever we make a modification in “**Angular Project**” so we need to copy the new files every time in to our **MVC Project Code** and to avoid this we can ask the compiler not to generate an **hash value** so that the file names will not be changing whenever we make modifications and to do that change the “**outputHashing**” attribute value to “**none**” in “**angular.json**” file which is “**all**” by default, as following:

Old Value => "outputHashing": "all"
New Value => "outputHashing": "none"

Now **save** the changes, go to **Command Prompt**, and then build the project using “**ng build --watch**” command as following:

```
<drive>:\<Personal Folder>\AngularInMVCCore\AngularInMVCCore\Angular\FirstProject> ng build --watch
```

Note: This will add a new folder under “**wwwroot**” folder with the name “**AngularScripts**” and this happens when we build the project for the first time.

Now add a new Controller in Controllers folder naming it as “**TestAngularController**” and add a **View** to the **Index** action method, delete the whole content in the file, and then under the “**AngularScripts**” folder we will find a file “**index.html**”, open it, copy the whole content in that file and paste it in to our “**Index.cshtml**” file. Now in the copied code under “**<body>**” tag we find 2 “**<script>**” tags that refers to the files “**polyfills.js**” and “**main.js**” of “**AngularScripts**” folder.

Right now, we copied the code of “**index.html**” present in “**AngularScripts**” to “**Index.cshtml**” file, which is present in a different folder, so we need to update the path of those scripts files for pointing them to “**AngularScripts**” folder as following:

Existing Code:

```
<script src="polyfills.js" defer></script> => <script src="~/AngularScripts/browser/polyfills.js" defer></script>
<script src="main.js" defer></script> => <script src="~/AngularScripts/browser/main.js" defer></script>
```

Modified Code:

Now run the “**TestAngular**” controller and invoke the “**Index**” action method which will execute the angular code and display’s the output here, so whenever we make changes in the “**Angular**” Project **re-build** it and run “**Index.cshtml**” again.

ASP.NET Razor Pages (Web Apps.)

The Razor Pages framework is lightweight and flexible framework which provides developers with full control over rendered HTML. Razor Pages is the recommended framework for cross-platform server-side HTML generation. Introduced as part of ASP.NET Core, is a server-side, page-focused framework that enables building dynamic, data-driven web sites with clean separation of concerns. Part of the ASP.NET Core web development framework from Microsoft, Razor Pages supports cross platform development and can be deployed to Windows, UNIX and Mac operating systems. Razor Pages makes use of the popular C# programming language for server-side programming, and the easy-to-learn Razor templating syntax for embedding C# in HTML mark-up to generate content for browsers dynamically.

Razor Pages is suitable for all kinds of developers from beginners to enterprise level. It is based on a page-centric development model, offering a familiarity to web developers with experience of other page-centric frameworks such as PHP, Classic ASP, Java Server Pages, and ASP.NET Web Forms. It is also relatively easy for the beginner to learn, and it includes all of the advanced features of ASP.NET Core (such as dependency injection) making it just as suitable for large, scalable, team-based projects.

Razor Pages is included within .NET Core from version 2.0 onwards, which is available as a free download as either an SDK (Software Development Kit) or a Runtime. The SDK includes the runtime and command line tools for creating .NET Core applications. The SDK is installed for you when you install Visual Studio 2017 Update 3 or later.

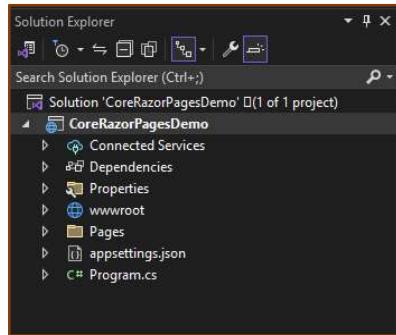
If you want a dynamic web site, that is one where the content is regularly being added to, you have a number of options available to you. You can use a Content Management System (CMS), of which there are many to choose from including Word Press, Umbraco, Joomla!, Drupal, Orchard CMS and so on. Or you can hire someone to build a suitable site for you. Or you can build your own if you have an interest in, and an aptitude for programming.

If you choose to build your own, you can choose from a wide range of programming languages and frameworks. If you are a beginner, you will probably want to start with a framework and language that is easy to learn, well supported and robust. If you are considering making a career as a programmer, you probably want to know that the skills you acquire while learning your new framework will enhance your value to potential employers. In both cases, learning C# as a language and ASP.NET Core as a framework will tick those boxes. If you are a seasoned developer, the Razor Pages framework is likely to add to your skillset with the minimum amount of effort.

You can still choose to use ASP.NET Core MVC to build your ASP.NET Core web applications. If you are porting an existing .NET Framework MVC application (MVC5 or earlier) to .NET Core, it may well be quicker or easier to keep with the MVC framework. However, Razor Pages removes a lot of the unnecessary ceremony that comes with the ASP.NET implementation of MVC and is a simpler, and therefore more maintainable development experience. The key difference between Razor Pages implementation of the MVC pattern and ASP.NET Core MVC is that Razor Pages uses the Page Controller pattern instead of the Front Controller pattern.

Creating a Razor Pages (Web App's.) Project: Open **Visual Studio** for creating a new project and in the **New Project** window select “**ASP.NET Core Web App**” project template, name it as “**CoreRazorPagesDemo**”, choose “**.NET 8.0 (Long Term Support)**” as **Framework**, select the Checkbox’s “**Configure for HTTPS**” and “**Do not use top-level statements**”, and click “**Create**” button.

If you now examine the project’s folder structure in **Solution Explorer** you will not find **Controllers**, **Models** and **Views** folder but you will find **Pages** folder where all the **Razor Pages** are stored. Rest of all the other folders and files will be same as ASP.NET Core MVC Project only:



Open “[Program.cs](#)” file and observe the code over there and we find the Razor Pages Service [enabled](#) here and the code in the class Program will be as below:

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.AddRazorPages();

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (!app.Environment.IsDevelopment())
        {
            app.UseExceptionHandler("/Error");
            // The default HSTS value is 30 days. You may want to change this for production scenarios, see
            // https://aka.ms/aspnetcore-hsts.
            app.UseHsts();
        }

        app.UseHttpsRedirection();

        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

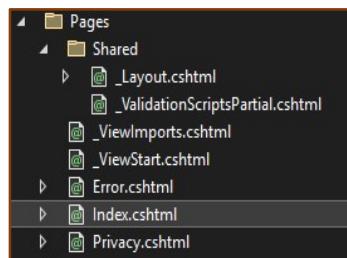
        app.MapRazorPages();

        app.Run();
    }
}
```

In the above code:

- AddRazorPages adds services for **Razor Pages** to the application.
- MapRazorPages adds **endpoints** for Razor Pages to the **IEndpointRouteBuilder**.

If we go to the Pages folder, we find a set of Razor Pages that are created by default, as below:



Basic code in a Razor Page will be as below:

```
@page
<h1>Hello, World!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

The above code looks a lot like a **Razor View** file used in an **ASP.NET Core MVC** application only. What makes it different is the **@page** directive. **@page** makes the file into an **MVC** action, which means that it handles requests directly, without going through a **Controller**. **@page** must be the first **Razor** directive on a **page**.

Every **Razor Page** will be having a **Page Model** class associated with it for implementing any business logic, for example if you have **Razor Page** with the name **Index** then the associated **Page Model** class will be "**IndexModel**" and this class inherits from the pre-defined class "**PageModel**" defined in **Microsoft.AspNetCore.Mvc.RazorPages** namespace. So, every **Razor Page** is associated with 2 files: "**Index.cshtml**" and "**Index.cshtml.cs**". Currently our project has a **Razor Page** with the name "**Index**" and the code present in associated 2 files will be as below:

Index.cshtml:

```
@page
@model IndexModel
 @{
    ViewData["Title"] = "Home page";
}
<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about
        <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core </a>.
    </p>
</div>
```

Index.cshtml.cs:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
namespace CoreRazorPagesDemo.Pages
{
```

```

public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;
    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }
    public void OnGet()
    {
    }
}

```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page. In the above code `page name` is “`Index`”, so `Page Model` class name is “`IndexModel`”. The `PageModel` class allows separation of the `logic` of a page from its `presentation`. It defines page handlers for requests sent to the page and the data used to render the page. The page has an `OnGet` handler method, which runs on `Get` requests and an `OnPost` or `OnPostAsync` handler method, which runs on `POST` requests (when a user posts the form). Handler methods for any `HTTP` verb can be added. The most common handlers are:

- `OnGet` to initialize state needed for the page. The `OnGet` method displays the `Index.cshtml Razor Page`.
- `OnPost` or `OnPostAsync` to handle form submissions. The `Async` naming suffix is optional but is often used by convention for asynchronous functions.

Note: If you’re familiar with `ASP.NET MVC` apps, The `OnPost` or `OnPostAsync` code looks similar to typical controller action method code. Most of the `MVC` primitives like `model binding`, `validation`, and `action results` work the same with `MVC` and `Razor Pages`.

Now let’s design the above application to perform CRUD Operations on our Customer table is MVCDB Database:

Step 1: To use `Scaffold Commands` and `Entity Framework Core` for `SQL Server` first we need to install the below packages using `NuGet Package Manager`:

```

Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.SqlServer

```

Step 2: Open `PMC (Package Manager Console)` and write the below code at the “`PM`” command prompt.

```

PM> Scaffold-DbContext "Data Source=Server;User Id=Sa;Password=123;Database=MVCDB;
TrustServerCertificate=True" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Customer

```

The above statement will add `Models` folder in the project and generate a `Model` class with the name “`Customer`” and also a `Context` class with the name “`MvcdbContext`”.

Note: Open “`Customer.cs`” file under which the `Customer` class is defined and change the `data type` of “`Status`” property from “`bool?`” to “`bool`” because in our `Database - Table`, `Status` column is applied with a `Not Null` constraint, so we want the `Status` property to be “`Non-Nullable`” but Entity Framework core defined it as “`Nullable`”.

```
(Old Code) public bool? Status { get; set; } => public bool Status { get; set; } (New Code)
```

Step 3: Now go to “**Startup**” class in case of **ASP.NET Core 5.0** or “**Program**” class in case of **ASP.NET Core 6.0** and **above** to register the “**DBContext**” class by calling “**AddDbContext**” method and to do this first we need to import the namespace “**CoreRazorPagesDemo.Models**” and write the below code.

ASP.NET Core 5.0: write the below statement under “**ConfigureServices**” method of “**Startup**” class:

```
services.AddDbContext<MvcdbContext>();
```

ASP.NET Core 6.0 and above: write the below statement under “**Program**” class just above the statement => “**var app = builder.Build();**”:

```
builder.Services.AddDbContext<MvcdbContext>();
```

Step 4: Create the required **Pages** for performing **CRUD** operations.

Display All Customers: Open the existing “**Index.cshtml.cs**” file, delete all the existing code inside of the “**IndexModel**” class and write the below code over there, by importing “**CoreRazorPagesDemo.Models**” namespace:

```
public class IndexModel : PageModel
{
    private readonly MvcdbContext context;
    public IndexModel(MvcdbContext context)
    {
        this.context = context;
    }
    public List<Customer>? Customers { get; set; }
    public void OnGet()
    {
        Customers = context.Customers.Where(C => C.Status == true).ToList();
    }
}
```

Now go to Index.cshtml and write the below code, by deleting the whole code in the page:

```
@page
@model IndexModel
 @{
    ViewData["Title"] = "Display Customers";
}
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" class="table table-bordered">
<tr>
<th>Custid</th>
<th>Name</th>
<th>Balance</th>
<th>City</th>
<th>Status</th>
<th>Actions</th>
</tr>
```

```

@foreach (var customer in Model.Customers)
{
<tr>
<td align="center">@customer.Custid</td>
<td>@customer.Name</td>
<td>@customer.Balance</td>
<td>@customer.City</td>
<td align="center"><input type="checkbox" asp-for="@customer.Status" disabled/></td>
<td>
<a asp-page="DisplayCustomer" asp-route-Custid="@customer.Custid">View</a> &ampnbsp
<a asp-page="EditCustomer" asp-route-Custid="@customer.Custid">Edit</a> &ampnbsp
<a asp-page="DeleteCustomer" asp-route-Custid="@customer.Custid">Delete</a>
</td>
</tr>
}
<tr>
<td colspan="6" align="center">
<a asp-page="AddCustomer">Add New Customer</a>
</td>
</tr>
</table>

```

Display Customer Page: Add a new - empty razor page under pages folder, name it as “`DisplayCustomer.cshtml`”. Go to “`DisplayCustomer.cshtml.cs`” view, import the namespace “`CoreRazorPagesDemo.Models`” and write the below code under the class by delete the existing code in the class:

```

public class DisplayCustomerModel : PageModel
{
    public Customer Customer { get; set; }
    private readonly MvcdbContext context;
    public DisplayCustomerModel(MvcdbContext context)
    {
        this.context = context;
    }
    public void OnGet(int Custid)
    {
        Customer = context.Customers.Find(Custid);
    }
}

```

Now go to `DisplayCustomer.cshtml` and write the below code, by deleting the whole code in the page:

```

@page "{Custid}"
@model CoreRazorPagesDemo.Pages.DisplayCustomerModel
 @{
    ViewData["Title"] = "Display Customer";
}

```

```

<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<table border="1" align="center" class="table-condensed">
<tr>
<td>Cusid:</td>
<td>@Model.Customer.Custid</td>
</tr>
<tr>
<td>Name:</td>
<td>@Model.Customer.Name</td>
</tr>
<tr>
<td>Balance:</td>
<td>@Model.Customer.Balance</td>
</tr>
<tr>
<td>City:</td>
<td>@Model.Customer.Custid</td>
</tr>
<tr>
<td>Status:</td>
<td><input type="checkbox" asp-for="@Model.Customer.Status" disabled/></td>
</tr>
<tr>
<td colspan="2" align="center">
<a asp-page="Index">Back to Customer Details</a>
</td>
</tr>
</table>

```

Add Customer Page: Add a new - empty razor page under pages folder, name it as “`AddCustomer.cshtml`”. Go to “`AddCustomer.cshtml.cs`” view, import the namespace “`CoreRazorPagesDemo.Models`” and write the below code under the class by delete the existing code in the class:

```

public class AddCustomerModel : PageModel
{
    public Customer Customer { get; set; }
    private readonly MvcdbContext context;
    public AddCustomerModel(MvcdbContext context)
    {
        this.context = context;
    }
    public RedirectResult OnPost(Customer customer)
    {
        context.Customers.Add(customer);
        context.SaveChanges();
        return Redirect("Index");
    }
}

```

Now go to AddCustomer.cshtml and write the below code, by deleting the whole code in the page:

```
@page
@model CoreRazorPagesDemo.Pages.AddCustomerModel
 @{
    ViewData["Title"] = "Add Customer";
}
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<form method="post">
<div>
    <label>Custid:</label><br /><input asp-for="@Model.Customer.Custid" />
</div>
<div>
    <label>Name:</label><br /><input asp-for="@Model.Customer.Name" />
</div>
<div>
    <label>Balance:</label><br /><input asp-for="@Model.Customer.Balance" />
</div>
<div>
    <label>City:</label><br /><input asp-for="@Model.Customer.City" />
</div>
<div>
    <label>Status:</label><br /><input asp-for="@Model.Customer.Status" />
</div>
<div><input type="submit" value="Save" /><input type="reset" value="Reset" /></div>
<div><a asp-page="Index" align="center">Back to Customer Details</a></div>
</form>
```

Edit Customer Page: Add a new - empty razor page under pages folder, name it as “EditCustomer.cshtml”. Go to “EditCustomer.cshtml.cs” view, import the namespace “CoreRazorPagesDemo.Models” and write the below code under the class by delete the existing code in the class:

```
public class EditCustomerModel : PageModel
{
    public Customer Customer { get; set; }
    private readonly MvcdbContext context;
    public EditCustomerModel(MvcdbContext context)
    {
        this.context = context;
    }
    public void OnGet(int Custid)
    {
        Customer = context.Customers.Find(Custid);
    }
    public RedirectResult OnPost(Customer customer)
    {
        customer.Status = true;
```

```

        context.Customers.Update(customer);
        context.SaveChanges();
        return Redirect("Index");
    }
}

```

Now go to EditCustomer.cshtml and write the below code, by deleting the whole code in the page:

```

@page "{Custid}"
@model CoreRazorPagesDemo.Pages.EditCustomerModel
 @{
    ViewData["Title"] = "Edit Customer";
}
<h2 style="text-align:center;background-color:yellowgreen;color:red">Customer Details</h2>
<form method="post">
<div>
    <label>Custid:</label><br /><input asp-for="@Model.Customer.Custid" readonly />
</div>
<div>
    <label>Name:</label><br /><input asp-for="@Model.Customer.Name" />
</div>
<div>
    <label>Balance:</label><br /><input asp-for="@Model.Customer.Balance" />
</div>
<div>
    <label>City:</label><br /><input asp-for="@Model.Customer.City" />
</div>
<div>
    <label>Status:</label><br /><input asp-for="@Model.Customer.Status" disabled />
</div>
<div>
    <input type="submit" value="Update" />
    <a asp-page="Index" align="center">Cancel</a>
</div>
</form>

```

Delete Customer Page: Add a new - empty razor page under pages folder, name it as “DeleteCustomer.cshtml”. Go to “DeleteCustomer.cshtml.cs” view, import the namespace “CoreRazorPagesDemo.Models” and write the below code under the class by delete the existing code in the class:

```

public class DeleteCustomerModel : PageModel
{
    public Customer Customer { get; set; }
    private readonly MvcdbContext context;
    public DeleteCustomerModel(MvcdbContext context)
    {
        this.context = context;
    }
}

```

```

    }
    public void OnGet(int Custid)
    {
        Customer = context.Customers.Find(Custid);
    }
    public RedirectToResult OnPost(Customer customer)
    {
        context.Customers.Update(customer);
        context.SaveChanges();
        return Redirect("Index");
    }
}

```

Now go to DeleteCustomer.cshtml and write the below code, by deleting the whole code in the page:

```

@page "{Custid}"
@model CoreRazorPagesDemo.Pages.DeleteCustomerModel
 @{
     ViewData["Title"] = "Delete Customer";
 }
<form method="post">
<div>
<label>Custid:</label><br />
<input asp-for="@Model.Customer.Custid" readonly />
</div>
<div>
<label>Name:</label><br />
<input asp-for="@Model.Customer.Name" readonly />
</div>
<div>
<label>Balance:</label><br />
<input asp-for="@Model.Customer.Balance" readonly />
</div>
<div>
<label>City:</label><br />
<input asp-for="@Model.Customer.City" readonly />
</div>
<div>
<label>Status:</label><br />
<input asp-for="@Model.Customer.Status" disabled />
</div>
<div>
<input type="submit" value="Delete" />
<a asp-page="Index" align="center">Cancel</a>
</div>
</form>

```

The End