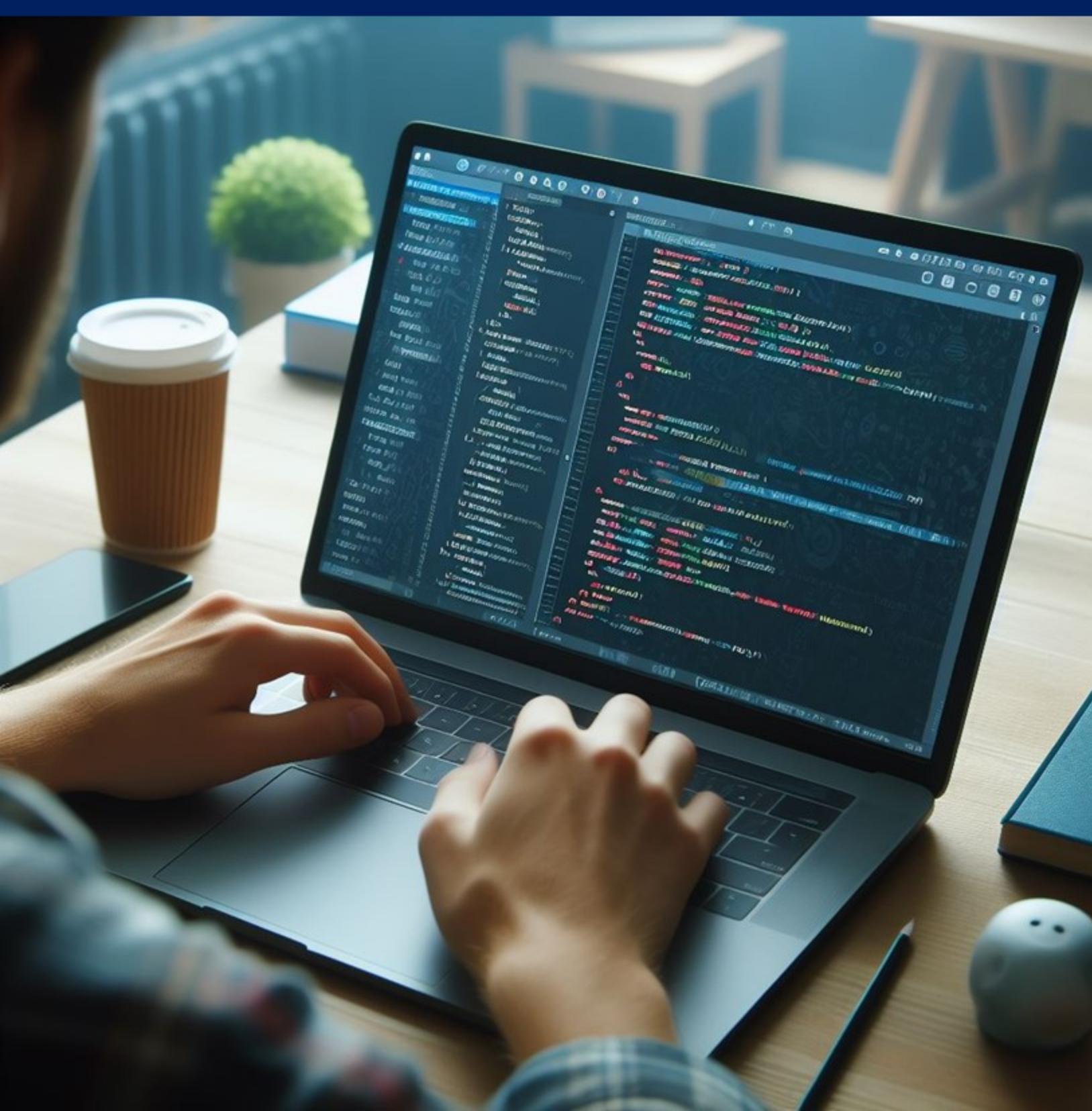


# Web Programming Using ASP.NET CORE and MVC



# Web Programming Using ASP.NET CORE and MVC

© 2024 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

**APTECH LIMITED**

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2024



Onlinevarsity



ASP.NET is a Web framework technology created by Microsoft for developing powerful, robust, and secure Web applications. ASP.NET Core is an open-source lightweight framework for building Web and cloud applications. ASP.NET MVC is a modern technology framework that enables developers to create powerful Web applications easily and efficiently. Microsoft Visual Studio 2022 includes templates and support for ASP.NET Web API, Core, MVC, and Core MVC applications.

This book embarks on a comprehensive journey through the landscape of ASP.NET and ASP.NET Core, delving into numerous topics essential for mastering Web development. Beginning with fundamental concepts such as Controllers, Views, and Models in ASP.NET MVC, it gradually progresses to cover advanced topics such as data handling, deployment patterns, and architectural principles. Additionally, concepts such as Onion Architecture, Kestrel Server, Fluent API, and Fluent Model in ASP.NET Core are thoroughly explained.

This book is the result of a concentrated effort by the Design Team, continuously striving to provide the best and latest in Information Technology. The design process adheres to the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's commitment to quality, our team conducts intensive research and curriculum enrichment to ensure alignment with industry trends.

We will be glad to receive your suggestions.

Design Team

## **Table of Contents**

- Session 1 – Introduction to ASP.NET and ASP.NET Core
- Session 2 – Working with ASP.NET Web Forms, Controls, and Events
- Session 3 – Working with ADO.NET and Entity Framework
- Session 4 – Client-side Development Using ASP.NET Core MVC
- Session 5 – More on ASP.NET MVC and Core MVC
- Session 6 – Action Methods and Advanced Concepts in MVC
- Session 7 – Enhancements in ASP.NET Core
- Session 8 – .NET Core Architecture and Kestrel Web Server Implementation
- Session 9 – Onion Architecture in ASP.NET Core – I
- Session 10 – Onion Architecture in ASP.NET Core - II
- Session 11 – Overview of Fluent Model and AutoMapper in ASP.NET Core MVC
- Session 12 – .NET Core Token Authentication in Web Applications
- Session 13 – Deployment and Unit of Work Patterns in ASP.NET Core
- Session 14 – User Login and ASP.NET Core Identity
- Session 15 – Publishing and Deploying ASP.NET Core Applications
- Appendix



**MANY  
COURSES  
ONE  
PLATFORM**



# Onlinevarsity App for Android devices

Download from **Google Play Store**

# Session 1: Introduction to ASP.NET and ASP.NET Core

## Session Overview

This session provides insights into ASP.NET and ASP.NET Core. It lists the features and advantages of ASP.NET and ASP.NET Core and explains how to choose between them.

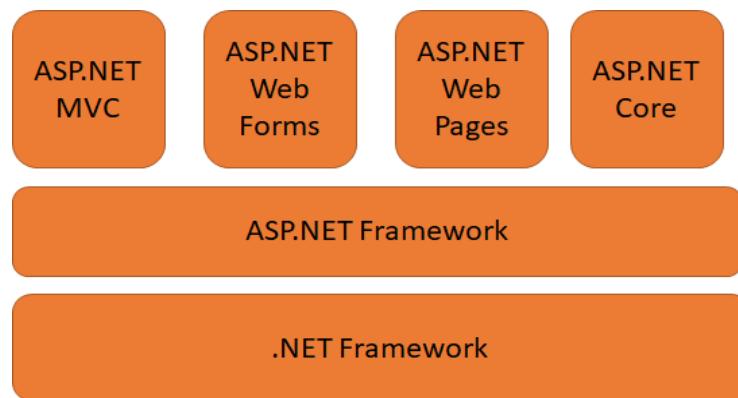
## Objectives

In this session, students will learn to:

- ✓ Explain ASP.NET Framework and its history
- ✓ Explain ASP.NET Page Lifecycle and Lifecycle Events
- ✓ Explain ASP.NET Features and its Uses
- ✓ Explain ASP.NET Core
- ✓ List ASP.NET Core advantages
- ✓ Identify how to choose between ASP.NET and ASP.NET Core
- ✓ Describe the features of ASP.NET Core 7.0

## 1.1 Introduction to ASP.NET

ASP.NET is a technology that is based on the .NET platform and is used to build dynamic Web applications. For developing Web applications, ASP.NET provides several frameworks such as Web Forms, ASP.NET MVC, ASP.NET Core, and ASP.NET Web Pages as shown in Figure 1.1. All these frameworks can be used to build enterprise-level Web applications.



**Figure 1.1: ASP.NET Frameworks**

Each framework caters to a distinct development style. The selection of framework depends on combination of programming assets in terms of knowledge and development experience.

The open-source version of ASP.NET, **ASP.NET Core**, runs on macOS, Linux, and Windows. It was first released in 2016 as a redesigned version of earlier Windows-only versions of ASP.NET.

### **1.1.2 History of ASP.NET**

During the year 1996, Microsoft introduced a new technology called ASP to help developers create Web applications. In those days, an ASP application included a Web page having VBScript or JScript server-side scripts connected to a server. When a request was received, the scripts were executed, and the corresponding HTML was generated.

### **Active Server Pages (ASP)**

Active Server Pages (ASP) was developed to generate Web content that could change based on the interaction with users. When a user requests an ASP page within HTML code, it is forwarded from the server hosting that page. The server then saves data regarding that user in the form of 'cookies'. This helps the server to make the content specific to that user. In Web development, this is called as 'dynamic content'.

A dynamic Web page shows different content every time a user views the page. Data is retrieved to generate the display and it is modified as per server functions or user constraints. For example, in case of a business Website, this approach is helpful for handling memberships, usernames, passwords, and similar data.

### **Evolution of ASP.NET**

In 2002, Microsoft released .NET Framework. With this release, the original version of ASP evolved into ASP.NET. In 2003, it was further upgraded and ASP.NET 1.1 was released. This paved the path for ASP.NET, which had many benefits over ASP. It was also a well-developed framework for Web application creation. Since then, ASP.NET has been developing with .NET Framework, with the major released one in 2005.

As the Web has been developing rapidly, Web pages are becoming more interactive and richer. In 2007, the next major release, ASP.NET 3.5, was delivered.

This release included major tools, such as ASP.NET AJAX, LINQ, and Dynamic Data. In 2009, ASP.NET 3.5 SP1 was released, which included a new MVC-based approach for processing Web page requests.

Simultaneously, when .NET Framework 4 was released, ASP.NET was also upgraded to version 4. The next crucial update was delivered together with the release of ASP.NET 4.5. Tools, such as Web API and SignalR were introduced.

The major and most vital event was the update and development of .NET Framework. In 2017, .NET Framework 4.7 was integrated with Windows 10 Creators Update. Next, version 4.8 was released in 2019. In May 2022, the version 4.8.0 was released. After this, version 4.8.1 was released. .NET Framework 4.8.1 is the last version of .NET Framework and no further versions may be released. This is because Microsoft has now rebranded the revamped platform to .NET.

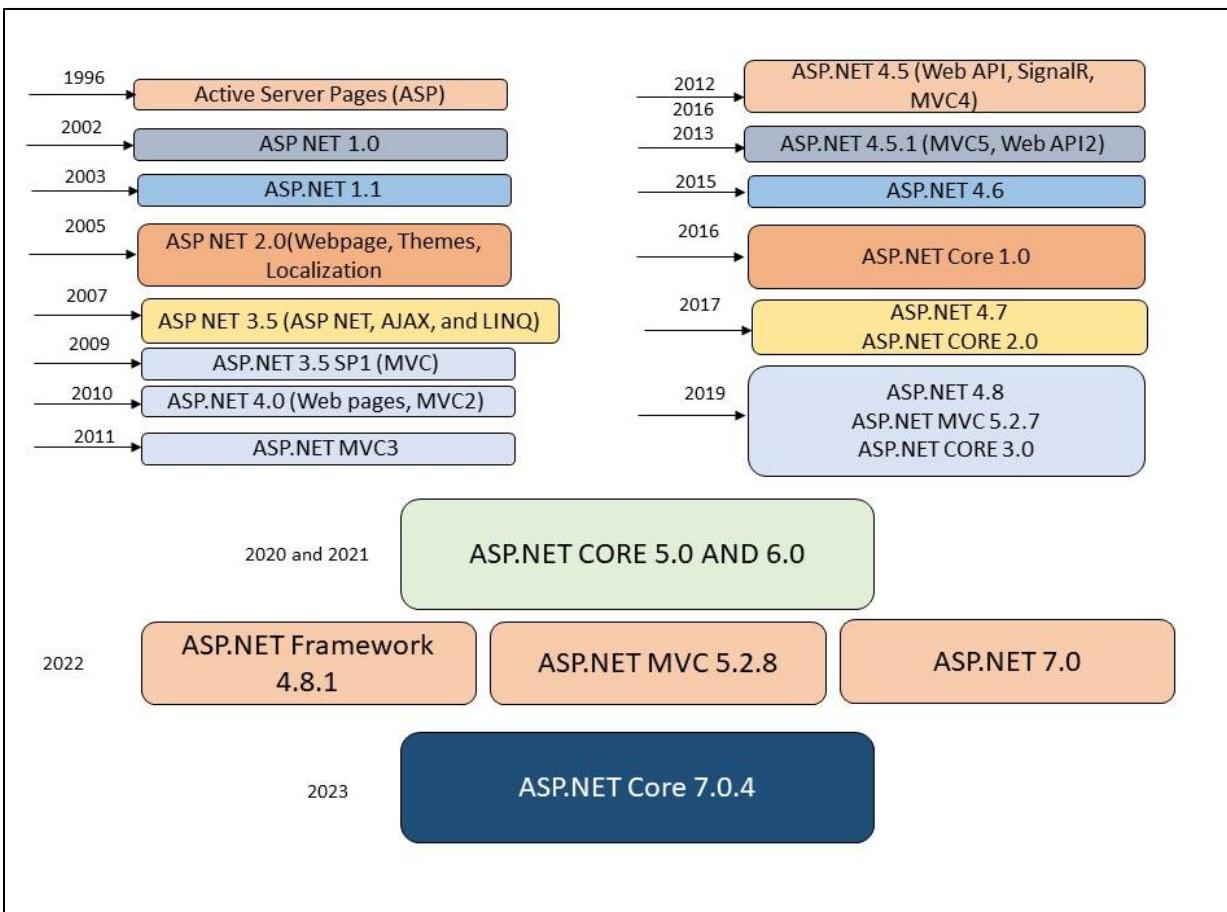
.NET is a free, open-source, cross-platform framework developed by Microsoft for building various types of applications, including Web, desktop, mobile, cloud, and IoT, using multiple programming languages such as C#, F#, and VB.NET.

Table 1.1 displays the recent versions of .NET and .NET Core.

Version	Release Date
.NET 7.0	November 8, 2022
.NET 6.0	November 8, 2021
.NET 5.0	November 10, 2020
.NET Core 3.1	December 3, 2019
.NET Core 3.0	September 23, 2019

**Table 1.1: Versions of .NET and .NET Core**

Figure 1.2 shows the release history of ASP.NET.



**Figure 1.2: Release History of ASP.NET, ASP.NET MVC, and Core**

## 1.2 Uses and Features of ASP.NET

Following are the uses of ASP.NET:

- It is an ideal approach to create multiple Web based applications according to the market trends. Complex applications can be created easily and both Web-based and desktop-based applications can be created in a much faster way.
- It offers versatile and dynamic library, which provide enhanced security.
- It considerably reduces the code in case of large Web applications development.
- It provides What You See Is What You Get (WYSIWYG) and offers server controls and blueprints having drag-and-drop facility and involuntary operation.
- It separates both HTML code and source code, thus, allowing any modifications to be performed easily.
- Static Web does not permit developers to write a code capable of executing when something specific happens.
- Though this can be achieved with the help of CGI scripts, it is lengthy and time consuming. However, with event handlers in ASP, this can be achieved without any flaws. For example, the 'Page\_Load' event that is initiated when a page is loaded or 'Unload', which is initiated the opposite way or in case the page is closed.
- With the 'runat' command, the ASP.NET framework makes sure that complete compatibility is achieved with various programming languages and also with pages of older versions of ASP.

Since ASP.NET belongs to the .NET platform, its code is managed code. ASP.NET applications are codes created with the help of components that can be extended and reused. These codes can utilize the complete hierarchy of classes available in the Framework.

ASP.NET provides several powerful features. Some of these are as follows:

### All-inclusive software infrastructure

It offers a programming model, an all-inclusive comprehensive infrastructure and numerous facilities necessary to create robust Web applications for computers and mobile devices.

### Abstraction Layer

It offers an abstraction layer atop the HyperText Transfer Protocol (HTTP) on which the Web applications are created. It offers advanced elements, such as classes and components within an object-oriented paradigm.

### Supports multiple languages

It allows the application to be coded in multiple languages, such as C#, Visual Basic.NET, Jscript, and J#.

### **Interactive data**

It is used to create collaborative, data-driven Web applications over the Internet. It includes various controls, for example, text boxes, buttons, and labels for building, setting up, and modifying code to generate HTML pages.

### **HTTP protocol**

It uses the HTTP protocol and with the help of HTTP commands and policies, permits interaction between a browser and the server.

### **Visual Studio**

Its application and front ends are created using Visual Studio.

Thus, it can be concluded that with the help of ASP.NET, developers can integrate the important elements of a business Website flawlessly.

## **1.3 ASP.NET Page Lifecycle**

The ASP.NET page lifecycle involves how ASP.NET deals with pages to generate results. It also dictates how the application and its related pages are instantiated and processed. The lifecycle can be categorized into following two types:

- Application lifecycle
- Page lifecycle

An ASP.NET page passes through a series of phases between its creation and disposal cycle. Initially, when a request for a page is made, the program loads the page into the server memory, processes it, and forwards it to the browser. Thereafter, it is dropped from the memory. At each of these stages, there are methods and events, which can be overridden depending on the application requirement. In simple words, developers can write their own code to replace the default code.

Following are the page lifecycle phases:

- Initialization
- Instantiation of page controls
- Recovery and upkeep of the state
- Execution of event handlers
- Rendering of page

It is important to understand the page cycle as it assists in coding for making a certain thing take place during any stage of the page lifecycle. It also assists in writing and initializing custom controls, filling in their properties with view-state information, and executing control behavior code.

Following are different stages of an ASP.NET page:

#### Requesting of page

When a page request is received, ASP.NET checks if it has to parse and compile the page, or if a cached version of the page exists. Based on this, it responds accordingly.

#### Starting of page lifecycle

The Request and Response objects are set. In case the request is not a new request or post back, then the IsPostBack property of the page is assigned true. The UICulture property is also set.

#### Initializing of page

The UniqueID property is set to allocate IDs to the controls on the page and the themes are applied. In case of a new request, it loads the postback data and the control properties are returned to the view-state values.

#### Loading of page

The control properties are set with the help of the view state and control state values.

#### Validating

Validate method available in the validation control is executed. Once successfully executed, the IsValid property of the page is set to true.

#### Postback event handling

In case the request is not a new request, the corresponding event handler is invoked.

#### Rendering of page

The view state and all controls for the page are stored. The page invokes the Render method for all the controls, the output of which is passed to the OutputStream class of the Response property of page.

#### Unloading

The rendered page is then forwarded to the client, the page properties are unloaded, and a complete cleanup is performed.

An ASP.NET application page comprises numerous server controls, which serve as the basic building blocks of the application.

Code Snippet 1 creates a sample page. Basic HTML controls are converted to server controls by adding "runat = server" and inserting an id attribute to assist in server-side processing.

The server when processing this code does not focus on controls, for example the header and anchor tags, and input elements. Instead, they are forwarded to the browser, which displays them.

### Code Snippet 1

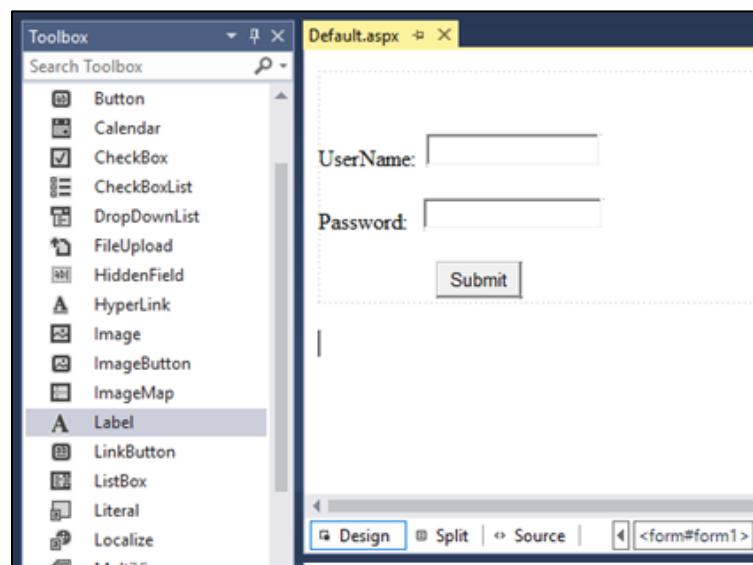
```
<head>
<style type="text/css">
    .auto-style1 {
        margin-right: 0px;
        margin-top: 7px;
    }
</style>
</head>
<form id="form1" runat="server">
<p>
    &nbsp;</p>
<p>
    &nbsp;<asp:Label ID="Label1" runat="server"
    Text="UserName"></asp:Label>
    :&nbsp;
<asp:TextBox ID="TextBox1" runat="server" CssClass=
    "auto-style1" Height="23px" Width="124px"></asp:TextBox>
</p>
<p>
    <asp:Label ID="Label2" runat="server"
    Text="Password"></asp:Label>
    :&nbsp;&nbsp;
<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
</p>
<p>
    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
    &nbsp;
    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
    <asp:Button ID="Button1" runat="server" Text="Submit" />
</p>
</form>
```

Enter the code in the **Source** tab of the Default.aspx file in the IDE. During actual development, it is recommended to design the form first using the **Toolbox** in the IDE rather than writing the code from scratch.

Click the **Design** option as shown in Figure 1.3 to view how the page looks.

**Figure 1.3: Page with Markup Showing the Design Tab**

The page shown in Figure 1.4 appears after clicking the **Design** tab. This is the Design editor for an ASP.NET page.

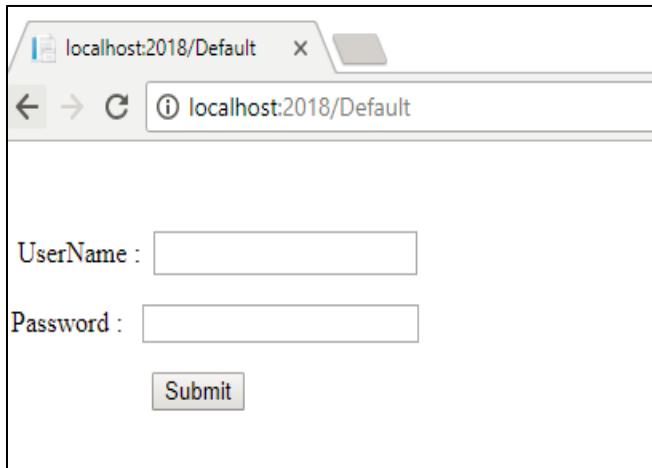


## Figure 1.4: Page Design

The left of the page shows the **Toolbox** section, which has many other HTML controls. To use these controls, just select the required control and drag and drop it on the page. After developers add controls, markup similar to Code Snippet 1 will be

generated in the Source tab.

The output shown in Figure 1.5 is displayed on running the page in the browser.



**Figure 1.5: Sample Output**

The lifecycle of an ASP.NET page is based on whether the page request is new or postback. Postback can be defined as the process using which a page places a request for itself.

Following is the lifecycle of a page when a new request is placed:

- **Initializing:** In this phase, the server generates a new instance of the server control.
- **Loading:** Here, the instance of the control generated in the preceding phase gets loaded onto the page object in which it is defined.
- **PreRendering:** Next, the control is updated to reflect the modifications done to it. This makes the control ready for rendering.
- **Saving:** In this phase, the state data related to the control is stored. For example, during the Load event, in case a value is fixed for the control, it is set in the HTML tag that will be sent back to the browser.
- **Rendering:** Here, the server generates the related HTML tag for the control.
- **Cleaning up:** In this phase, all cleanup activities, for example, terminating files and database connections started by the control are done.
- **Unloading:** Lastly, all cleanup activities, for example, terminating the instances of server control are done. This concludes the lifecycle of a server control.

In case of a postback event, the lifecycle of a page is as follows:

- **Initializing:** The server control's new instance comes into action.
- **Loading view state:** The control's view state loads again into the new instance. The former control, whose view state is loaded, is the one that the client has sent.
- **Loading:** The control's instance is added to the page object. It is in this object

that the instance is described.

- **Loading the postback data:** In this phase, the server looks up for information related to the control that is loaded in client-sent data.
- **PreRendering:** Updates are made to the control to reflect the modifications done to it. This makes the control ready for rendering or execution.
- **Saving state:** The variation in the control's state between the present and the preceding request of the page is stored. For each variation, the related event is triggered. For instance, in case the textbox is modified, the latest text is stored while triggering `text_change` event.
- **Rendering:** Here, the server generates the control's related HTML tag.**Disposing:** All activities that the control started are terminated, such as closing a database connection.
- **Unloading:** Lastly, all pending tasks are terminated, such as deleting the server control.

Following are the events connected with the relevant page cycle phases:

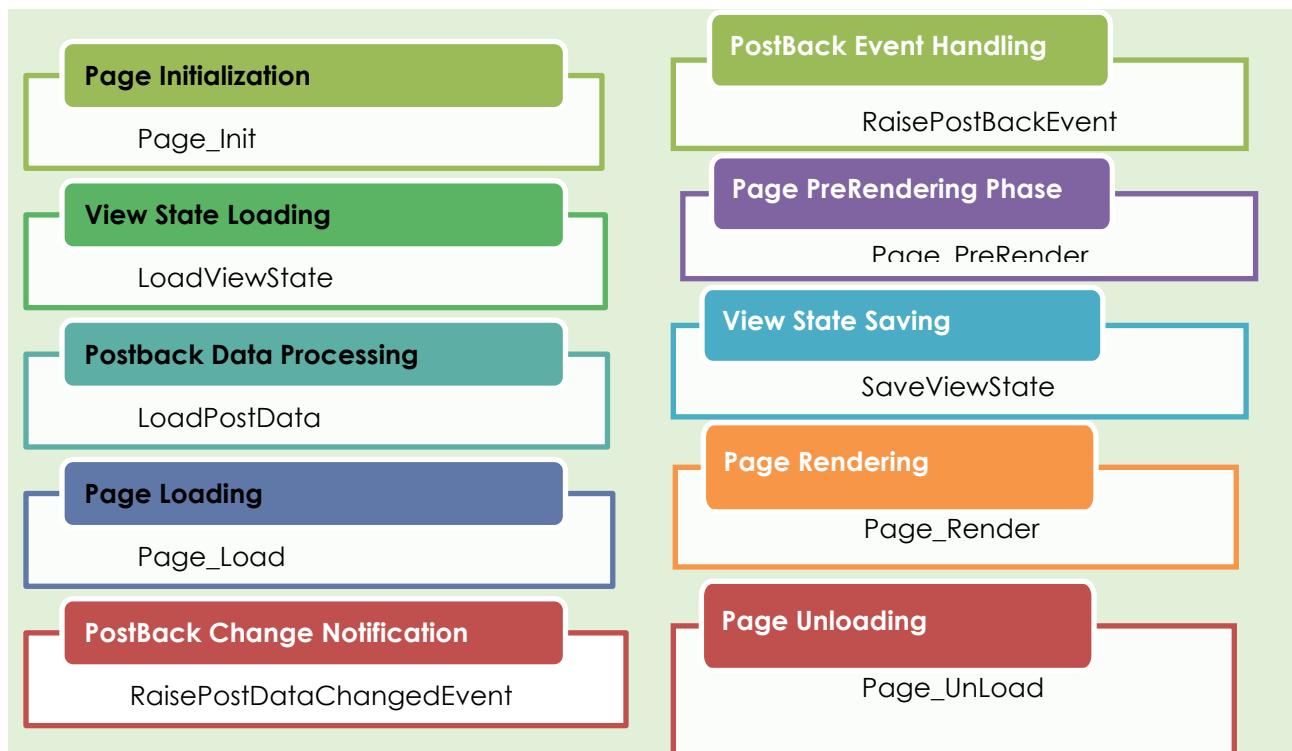
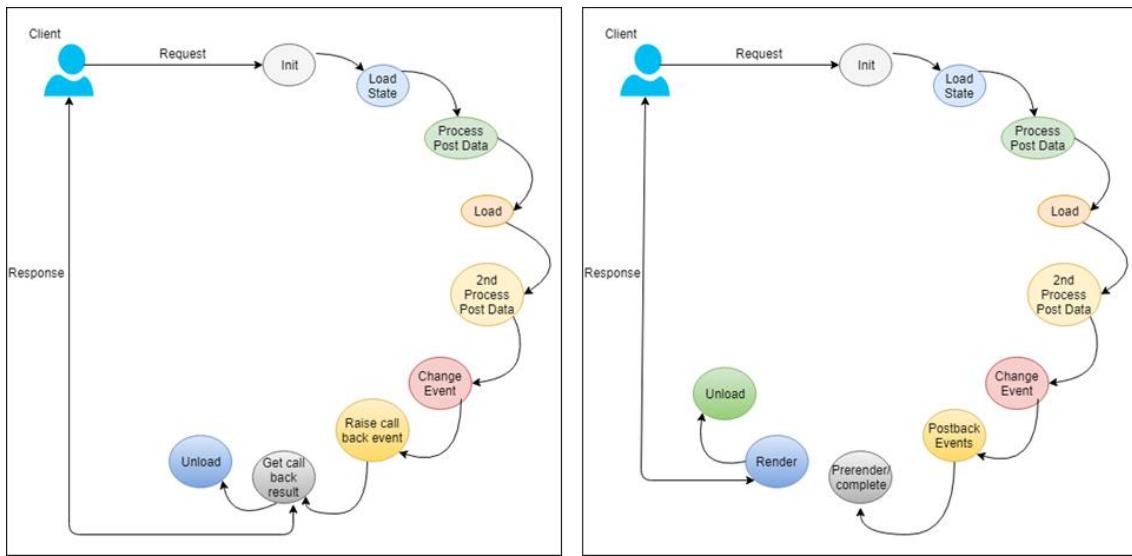


Figure 1.6 shows how the server processes the controls on an ASP.NET page. The image on the left shows the controls in case of postback and the one on the right shows callback.



**Figure 1.6: Processing of Controls (Postback and Callback)**

## 1.4 ASP.NET Core Introduction

ASP.NET Core is a new open-source and cross-platform framework. It facilitates easier development of modern Web applications, including cloud-based applications. Some examples are Web apps, IoT apps, and mobile backends.

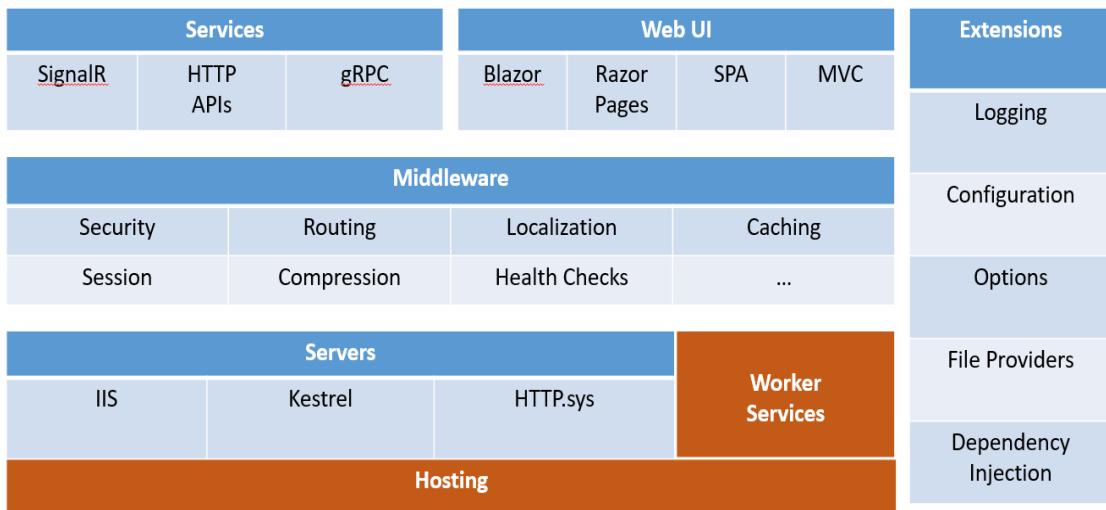
ASP.NET Core applications have the ability to execute on both .NET Core and .NET Framework. As it involves modular components having minimum overhead, developers can retain flexibility when creating solutions. They can create and execute cross-platform ASP.NET Core apps on Windows, Mac OS, and Linux.

ASP.NET Core is not dependent on `System.Web.dll` any longer. It is now supported by NuGet packages. This helps developers to optimize their app to take in only the NuGet packages they require. Advantages of a lesser app surface area are enhanced security, decreased servicing, enriched performance, and reduced expenditure.

Following are the improvements offered by ASP.NET Core:

- Streamlined Web development
- A system that is set to work on cloud
- Good community base
- An integrated platform for creating a variety of Web applications and APIs
- Assimilation of latest frameworks that work on client computers and development workflows
- Support for a flexible and lightweight HTTP request channel that ensures optimal performance
- Support for hosting itself in a targeted process or on different platforms, such as Docker and Apache
- Simultaneous versioning of applications

Figure 1.7 depicts an overview of ASP.NET Core.



**Figure 1.7: Comprehensive Web Development with ASP.NET Core**

## 1.5 ASP.NET Core Advantages

Following are the advantages of ASP.NET Core:

It supports Dependency Injection (DI)

DI promotes a successful unit testing as it separates resolution type and lifetime management. It is available as an additional feature as the proprietary or a third-party library. Thus, it is now a part of the framework.

It provides cross-platform compatibility

ASP.NET Core 1.0 was created in a way to utilize the new .NET Core 1.0. The code is included with the deployed app, which works even on OSX and Linux operating systems. For developers who prefer to work on any of these platforms, there is a tool called Visual Studio Code editor. It helps them to create applications on any of these platforms.

It has simplified MVC and WEB.API development

Previously, MVC and Web API were dependent on dissimilar versions of the framework: `System.Web.Mvc` and `System.Web.Http`/`System.Net.Http`, respectively. This created confusion. Well, this is now fixed in MVC 6 onwards, which offers them in one namespace, `Microsoft.AspNet.Mvc`.

It focuses on increasing productivity

Microsoft offers Kestrel, a lightweight Web server offering numerous benefits, including cross-platform compatibility.

#### It offers an open-source environment

GitHub provides all the source codes for ASP.NET Core 1.0 and higher version packages. It offers numerous repositories, which hold codes for important functionalities of ASP.NET, along with demos. This not only helps in speeding up the development cycle, but also ensures continued code enhancement.

#### It promotes modularity

From VS 2010 onwards, developers are utilizing both the Package Manager Console and the NuGet Package Manager for setting up frameworks and libraries. Now, the same tools are allowing them to use the ASP.NET features for their projects. This helps them to pay more attention to the libraries that are used in their solutions.

From these advantages, it can be noted that ASP.NET on the .NET Core framework offers the most adaptable version of ASP.NET yet. Its distinct features, such as modular design, restructured programming model, and efficiency developments will definitely make it well accepted by developers.

## 1.6 Choosing between ASP.NET and ASP.NET Core

While ASP.NET is an established framework that offers all the elements necessary to develop enterprise-grade, server-based Web apps on Windows, ASP.NET Core is an open-source framework that helps in developing latest, cloud-based Web apps on not just Windows operating system, but also on MacOS and Linux.

Table 1.2 will help in deciding whether ASP.NET or ASP.NET Core is better to suit the requirements.

ASP.NET Core	ASP.NET
Apps can be built either for Windows, MacOS, or Linux.	Apps can be built only for Windows.
It is recommended when developing a Web UI as of ASP.NET Core 2.x is Razor Pages.	It is recommended when developing a Web UI use Web Forms, SignalR, MVC, Web API, or Web pages.
Many versions can be utilized per machine.	Only one version can be utilized per machine.
Apps can be created with the help of Visual Studio, Visual Studio for Mac, or Visual Studio Code using C# or F#.	Apps can be created with the help of Visual Studio using C#, VB, or F#.
Performance is better than ASP.NET.	Performance is good, however, less than ASP.NET Core.
Developers can choose either .NET Framework or .NET Core runtime.	Developers must use .NET Framework runtime.

**Table 1.2: Difference between ASP.NET or ASP.NET Core**

## 1.7 Blazor

Blazor is a .NET Web framework used for developing client-side applications in C#/Razor and HTML as shown in Figure 1.7. Applications are run in the browser using

Web Assembly. Web Assembly is low-level assembly-like language with a compact binary format that is compatible with modern Web browsers for execution.

Blazor can simplify the process of developing Single-Page Applications (SPAs) while also enabling full-stack Web development with .NET. It also extends rendering of UI as HTML and CSS to support a wide range of browsers including mobile browsers.

Finally, it allows integration with modern hosting platforms such as Docker. Blazor also increases the flexibility of using .NET for client-side Web development by sharing app logic across client and server. It takes advantage of performance, reliability, and security. Visual Studio can be used on Windows, Linux, and macOS to remain productive. It is built on a shared set of languages, frameworks, and tools that are stable, feature-rich, and simple to use.

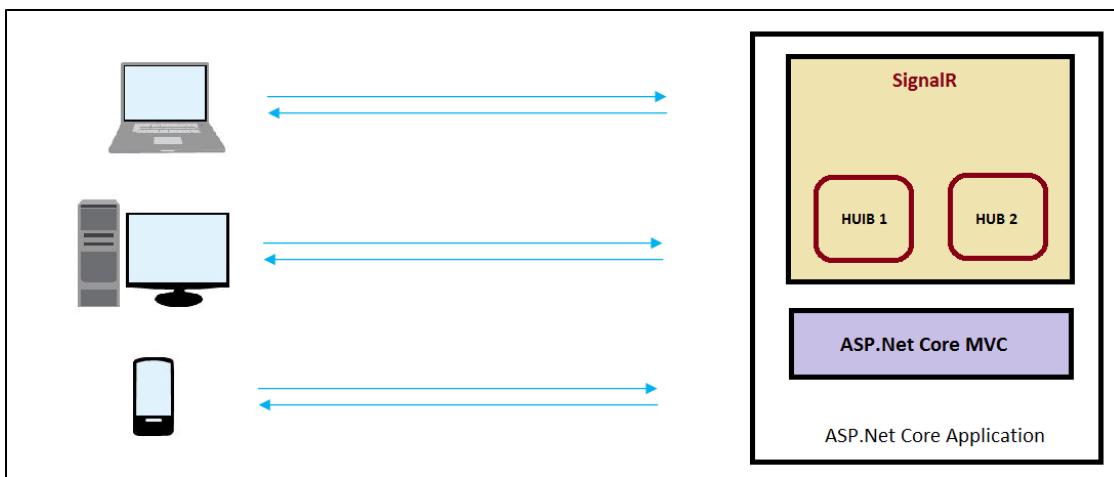
## 1.8 SignalR

Real-time Web applications enable users to receive the most up-to-date information, such as messages, alerts, or application/user-specific data, as soon as the data changes (add/modify/delete) on the server without the user having to request the data.

SignalR is an open-source library that enables ASP.NET developers to easily integrate real-time functionality into enterprise Web applications. It allows real time pushing of information to clients simultaneously. SignalR Core has been available in ASP.NET Core since v2.1 as a cross-platform solution for adding real-time features to Web applications and other applications. Some examples of applications are as follows:

- Applications that require frequent server updates, such as games, social networks, shares, and markets
- Applications in organizations where there is a dashboard that monitor sales and inventory.
- Applications that provide real time team communication.
- Applications that require notifications for users for every update.

SignalR includes an API for making server-to-client Remote Procedure Calls (RPC). RPCs call functions on clients using server-side .NET Core code. There are several platforms that are supported each with its own client SDK. As a result, the programming language invoked by the RPC call varies. Figure 1.8 displays the working of SignalR.



**Figure 1.8: Working of SignalR**

## 1.9 Dapper

Dapper is a straightforward Object Mapper that performs Object-Relational Mapping (ORM) and bridges the mapping between databases and programming languages. It is also known as the Micro ORM. It is comparable in speed to an ADO.NET data reader and Entity Framework.

The `QueryAsync()` method as shown in Code Snippet 2 is a Dapper extension method. It takes the query and required parameters object to build the command. It returns objects of the specified type for further processing. There are also numerous methods for returning a single object or the results of multiple queries sent to the server in a single batch.

### Code Snippet 2

```
public async Task<IEnumerable<Customer>> ListCustomers(int pg,
int pgSize){
    using (var connection = _provider.GetDbConnection())
    {
        var parameters = new { Skip = (pg - 1)*pgSize, Take =
pgSize };
        var sql_query = "SELECT * FROM Customers ORDER By
CURRENT_TIMESTAMP ";
        return await
connection.QueryAsync<Customer>( sql_query, parameters);
    }
}
```

## 1.10 Features of ASP.NET Core 7.0

The most recent release of ASP.NET Core is version 7.0.

It was officially launched on November 8, 2022. To develop Web applications using ASP.NET Core 7.0, it is essential to use Visual Studio 2022 version 17.4.0 or later.

Key highlights of the ASP.NET Core 7.0 update include several noteworthy features and improvements such as:

- **Servers and Runtime**

Developers or administrators can:

- Control the rate of processed requests through adaptable endpoint settings and policies.
- Set up response caching to optimize request handling.
- Handle requests with compressed content.
- Integrate native support for HTTP/3, the most recent version of HTTP utilizing the new Quick UDP Internet Connections (QUIC) multiplexed transport protocol.
- Utilize WebSockets on HTTP/2 connections.
- Generate streams and datagrams over HTTP/3, with experimental backing for WebTransport.

- **Minimal APIs**

Developers or administrators can:

- Employ endpoint filters to execute cross-cutting code before or after a route handler.
- Provide strongly typed results from minimal APIs.
- Arrange groups of endpoints with a shared prefix.

- **gRPC**

Developers or administrators can:

- Broaden the accessibility of your gRPC services by exposing them as JSON-based APIs.
- Experiment with generating OpenAPI specifications for your gRPC JSON transcoded services.
- Monitor and assess the health of gRPC server applications.

- **SignalR**

- Respond to server requests with client results.

- **MVC**

- Enhance the experience of null state checking by supporting nullable page and view models.

- **Blazor**

- Optimize Blazor performance results in a faster Blazor WebAssembly.

# Summary

- ✓ Active Server Pages (ASP) was developed with an aim to generate Web content capable of changing based on the interaction with the user.
- ✓ With the help of ASP.NET, developers can integrate important elements of a business Website flawlessly and with a simple code.
- ✓ The ASP.NET lifecycle can be categorized into Application Lifecycle and Page Lifecycle.
- ✓ The lifecycle of an ASP.NET page is based on whether the page request is new or a postback.
- ✓ ASP.NET Core is a new open-source and cross-platform framework that helps developers to create novel cloud-based Internet associated applications.
- ✓ ASP.NET is an established framework that offers all the elements necessary to develop enterprise-grade, server-based Web apps on Windows.
- ✓ ASP.NET Core helps in developing apps on not just Windows operating system, but also on MacOS and Linux.
- ✓ ASP.NET collaborates with popular JavaScript frameworks.
- ✓ Blazor is a .NET Web framework used for developing client-side applications in C#/Razor and HTML.
- ✓ Dapper is a straightforward Object Mapper that performs Object-Relational Mapping (ORM).
- ✓ Features and improvements introduced in ASP.NET Core 7.0, including enhanced Servers and Runtime, streamlined development with Minimal APIs, improved gRPC services accessibility, advanced SignalR capabilities, MVC enhancements.

# Test Your Knowledge



1. In which of the following stages of an ASP.NET page, the Request and Response objects are set?

<b>A</b>	Page initialization
<b>B</b>	Page load
<b>C</b>	Starting of page lifecycle
<b>D</b>	Page request

2. Which of the following event is connected with the PostBack Event Handling phase?

<b>A</b>	RaisePostDataChangedEvent
<b>B</b>	RaisePostBackEvent
<b>C</b>	Page_PreRender
<b>D</b>	LoadpostData

3. In the lifecycle of a page when a new request is placed, the server generates the related HTML tag for the control in which of the following stages?

<b>A</b>	Rendering
<b>B</b>	PreRendering
<b>C</b>	Disposing
<b>D</b>	Loading

4. From \_\_\_\_\_ onwards, developers can utilize both the Package Manager Console and the NuGet Package Manager for installing and configuring frameworks and libraries.

<b>A</b>	Visual Studio 2011
<b>B</b>	Visual Studio 2009
<b>C</b>	Visual Studio 2012
<b>D</b>	Visual Studio 2010

5. ASP.NET Core allows the application to be coded in languages such as C#, Visual Basic.NET, Jscript, and J#. (State True/False)

<b>A</b>	True
<b>B</b>	False

## Answers

1	C
2	B
3	A
4	D
5	B

## **Try It Yourself**

1. Create a simple ASP.NET Web Form using Visual Studio 2022:
  - I. Use Visual Studio to create a new ASP.NET Web Forms application.
  - II. Design a simple form with labels, textboxes, and a submit button.
  - III. Handle the button click event to display a message using server-side code.
2. Explore Page Lifecycle in the application:
  - I. Investigate the ASP.NET page lifecycle by adding debugging statements in various lifecycle events.
  - II. Use breakpoints to observe how the page progresses through initialization, rendering, and disposal.

# *Session 2: Working with ASP.NET Web Forms, Controls, and Events*

## **Session Overview**

This session outlines fundamentals of Web application development and Web Forms. The session also covers various kinds of controls in ASP.NET.

## **Objectives**

In this session, students will learn to:

- ✓ Explain Web application development and Web Forms
- ✓ Identify types of Web Forms
- ✓ Describe event handling in ASP.NET
- ✓ List and describe various types of controls in ASP.NET
- ✓ Explain Web API and Web security

## **2.1 Introduction to Web Application Development and Web Forms**

A Web application is composed of code pages and documents in different formats. The basic type of document may be a static HTML page, which contains data that could be formatted and displayed by an online browser. An HTML page may additionally contain hyperlinks to alternative HTML pages. Wherever the target document is located, that location is referred to as a hyperlink or Uniform Resource Locator (URL). The ensuing combination of content and links is usually referred to as machine-readable text and provides straightforward navigation to an enormous quantity of data on the World Wide Web.

A Web application is a computer program located on a remote server and passed over the network such as a browser application. Web applications are built with online forms, word processors, shopping carts, video and photo editing, file scanning,

spreadsheets, file conversions, and email programs such as Gmail.

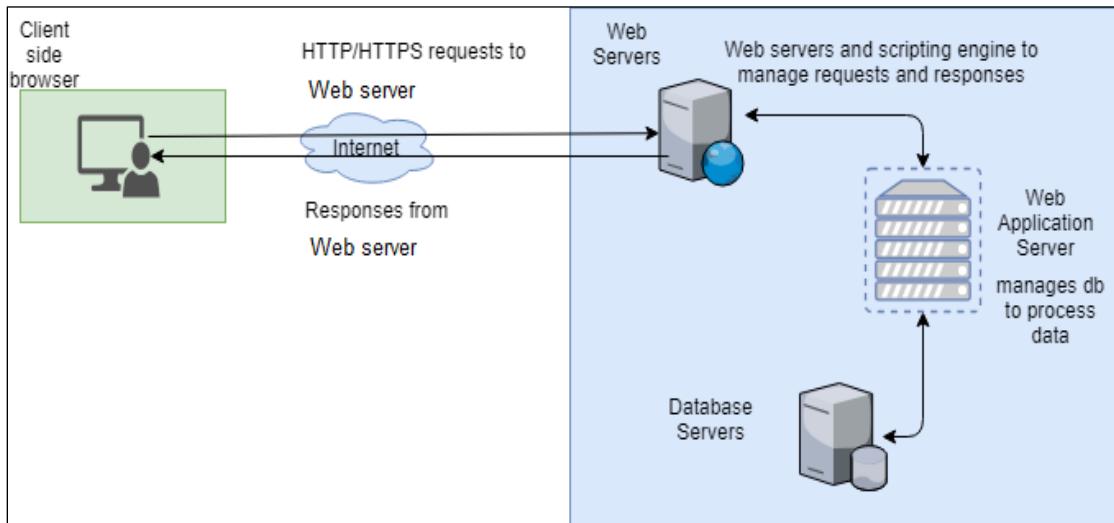
ASP.NET is the advanced technology for Web development. It adds various new features and picks out the best from Active Server Pages (ASP) along with the services and features from Common Language Runtime (CLR). This provides a fast Web development experience with lesser code.

ASP.NET Websites are created using ASP.NET technology with .NET Framework. The three programming models that are used for creating ASP.NET Websites are namely, ASP.NET Web Forms, ASP.NET Web Pages, and ASP.NET Model-View-Controller (MVC).

ASP.NET Web Forms are Web pages that are returned when a user requests for an action from Web browsers. The requested Web page is applied on the servers and results in the shape of integrated elements in a clean design with data providing the structure to the Web applications.

The request to the Web application server is initiated by the user through browser using database servers to execute the desired task such as updating and fetching the data. The browser then displays the requested information to the user through the Web application.

Figure 2.1 depicts the Web application development model.



**Figure 2.1: Web Application Process**

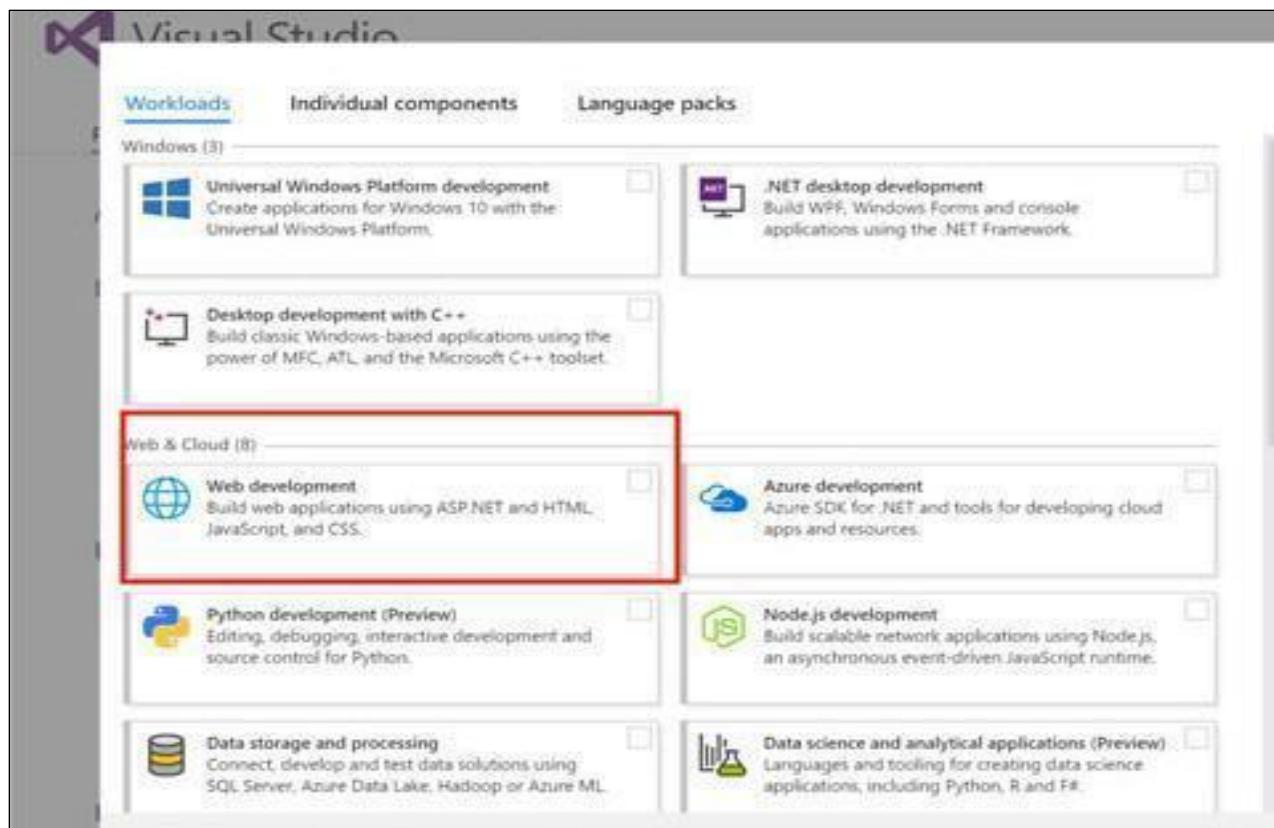
## 2.2 ASP.NET Environment Setup

Microsoft recommends developing ASP.NET Web applications in an isolated development environment. The key development tool for building ASP.NET applications and front end is Visual Studio. It is an Integrated Development Environment (IDE) for writing, compiling, and debugging code.

It provides a complete set of development tools for building ASP.NET Web applications, Web services, desktop applications, and mobile applications. To create ASP.NET Web Forms and other Web applications, Visual Studio 2022 provides an exceptionally capable IDE.

At the time of installation, it is essential one selects the appropriate workloads to ensure correct templates.

Refer to Figure 2.2.



**Figure 2.2: Selecting Workloads for Web Development**

## 2.3 ASP.NET Web Forms

Web Forms are Web pages that provide an interactive user interface with the help of various controls. The lifecycle of Web Forms is similar to the lifecycle of Web processes that run on the server. When a Web Forms page is processed, the processing information is passed to the browser using the HTTP protocol.

To understand how a Web Forms page works in Web applications, developers must learn about what goes inside a Web page when it is being processed.

### Processing of an ASP.NET Page

All ASP.NET pages in a Web application are compiled on the server. The compilation of the ASP.NET page starts when the URL is requested by the user for the first time. When a user requests a URL, the ASP.NET code is sent to the ASP.NET script engine in Internet Information Services (IIS). The script engine then generates the ASP.NET controls as well as the Web Forms page, which is then displayed as the URL response in the Web browser of the client machine.

A Web Form page is made up of two segments, which are as follows:

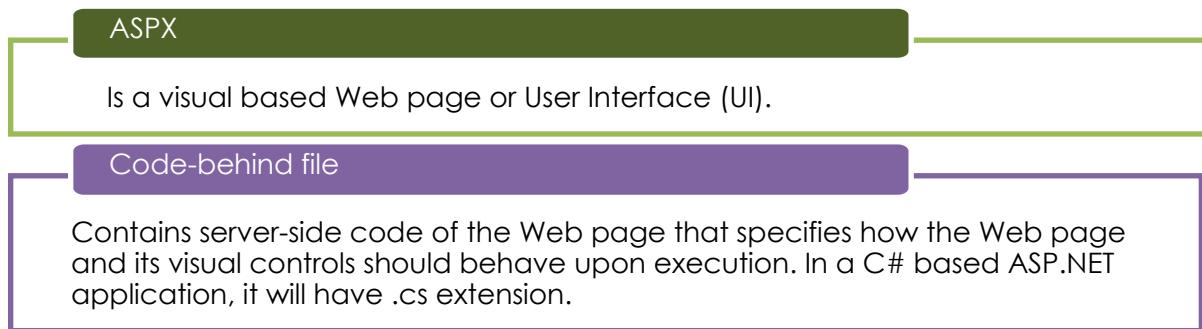
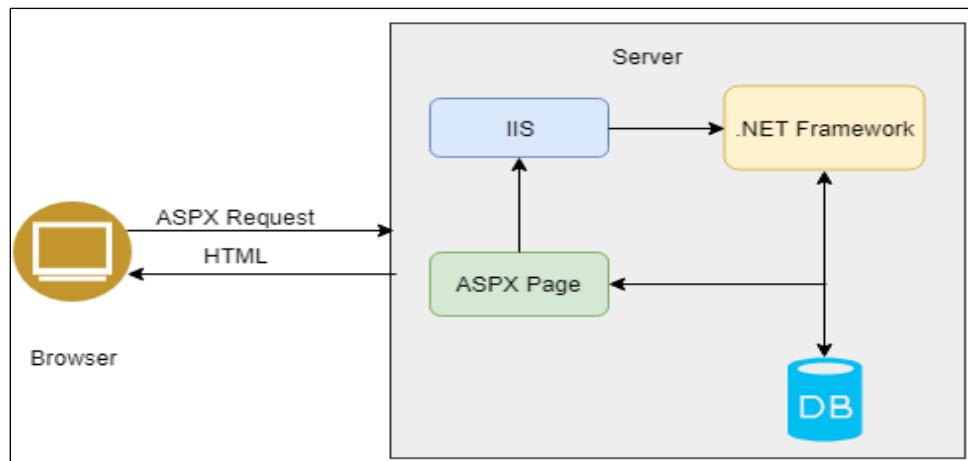


Figure 2.3 depicts how a Web Form works.

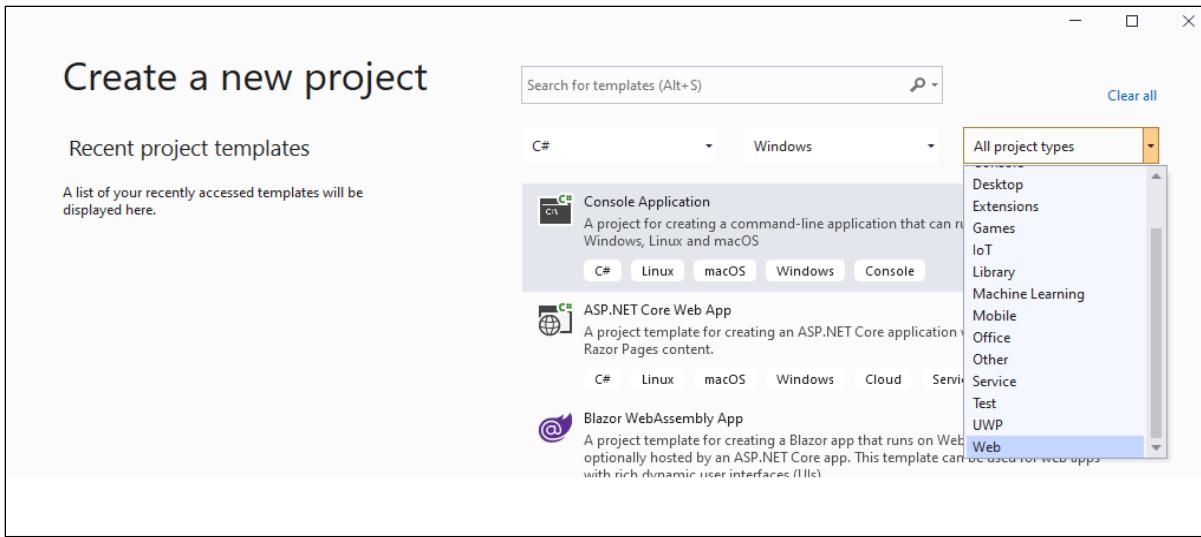


**Figure 2.3: ASP.NET Web Form**

### 2.3.1 Creating an ASP.NET Web Form

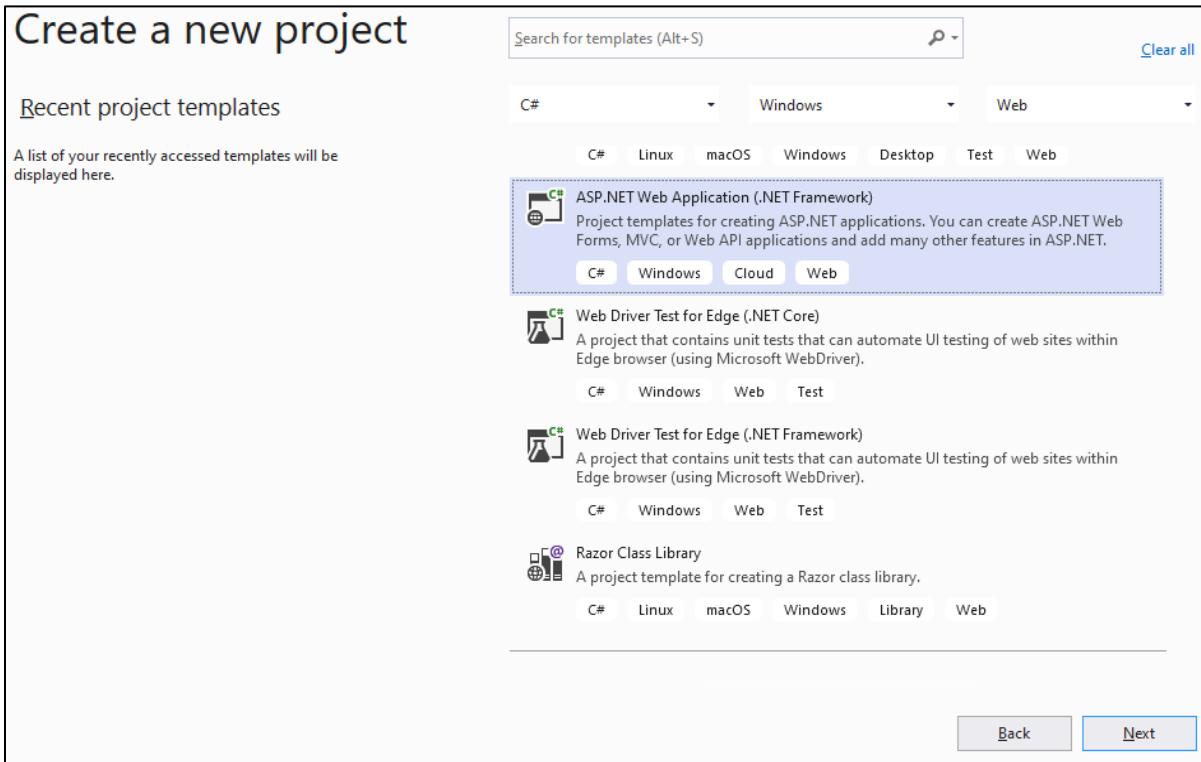
To create a Web Form using the IDE, open Visual Studio 2022. Then, perform following steps:

1. Click **Create a new project** option. In the Create a **New Project** window, select option **Web** from All Project Types drop-down list as shown in Figure 2.4.



**Figure 2.4: Create a New Project**

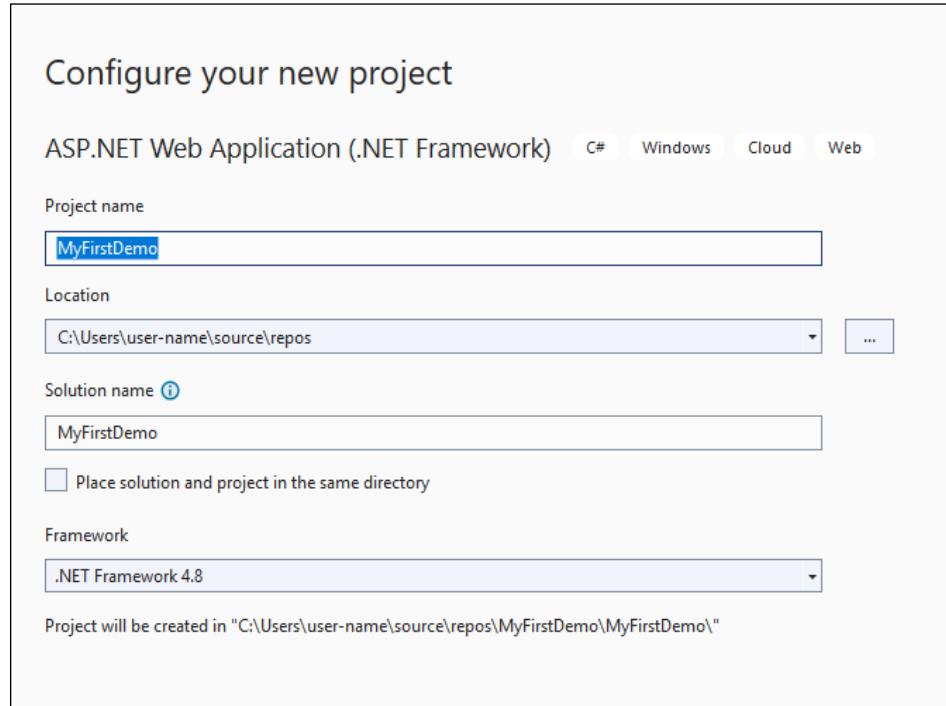
2. Select **ASP.NET Web Application (.NET Framework)** and click **Next** as shown in Figure 2.5.



**Figure 2.5: ASP.NET Web Application**

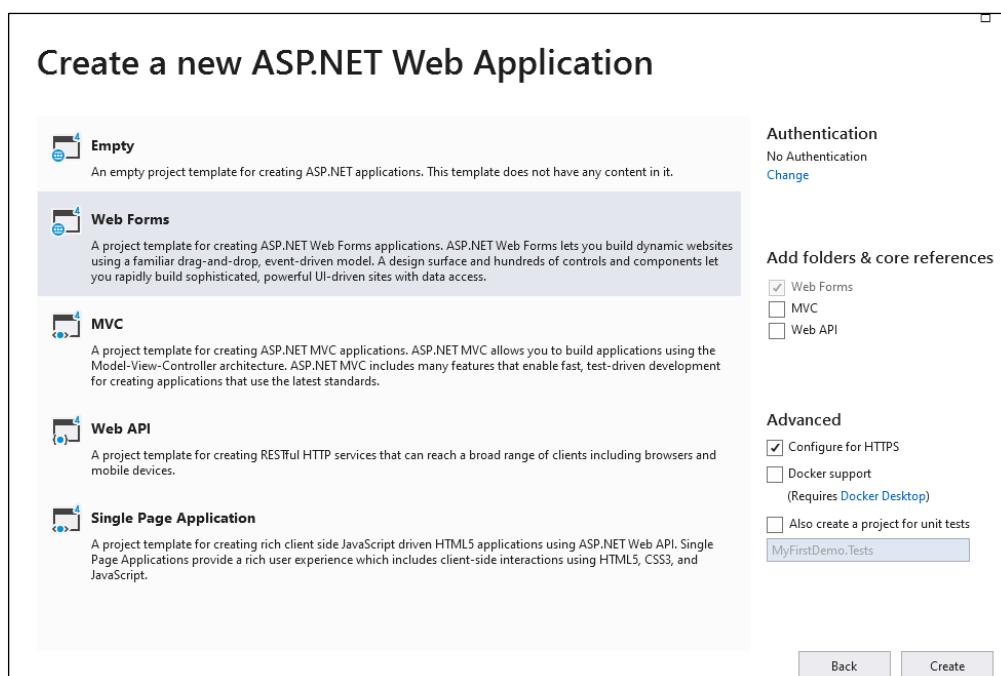
Note: If this option is not visible, it means that .NET framework development tools was not selected during installation. You can modify the Visual Studio 2022 installation and ensure the option is selected.

3. Specify the name **MyFirstDemo** and appropriate location for the project and click **Create**, as shown in Figure 2.6.



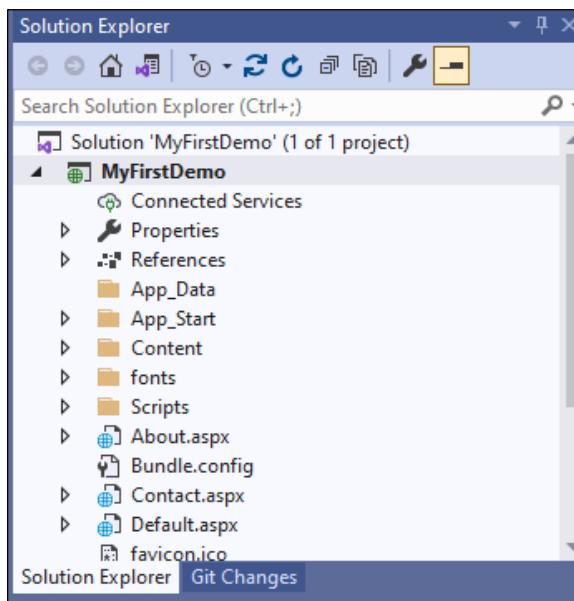
**Figure 2.6: Configure Your New Project Window**

4. Next, select the type of application to be created. Select the **Web Forms** option as shown in Figure 2.7.



**Figure 2.7: Selecting a Web Form**

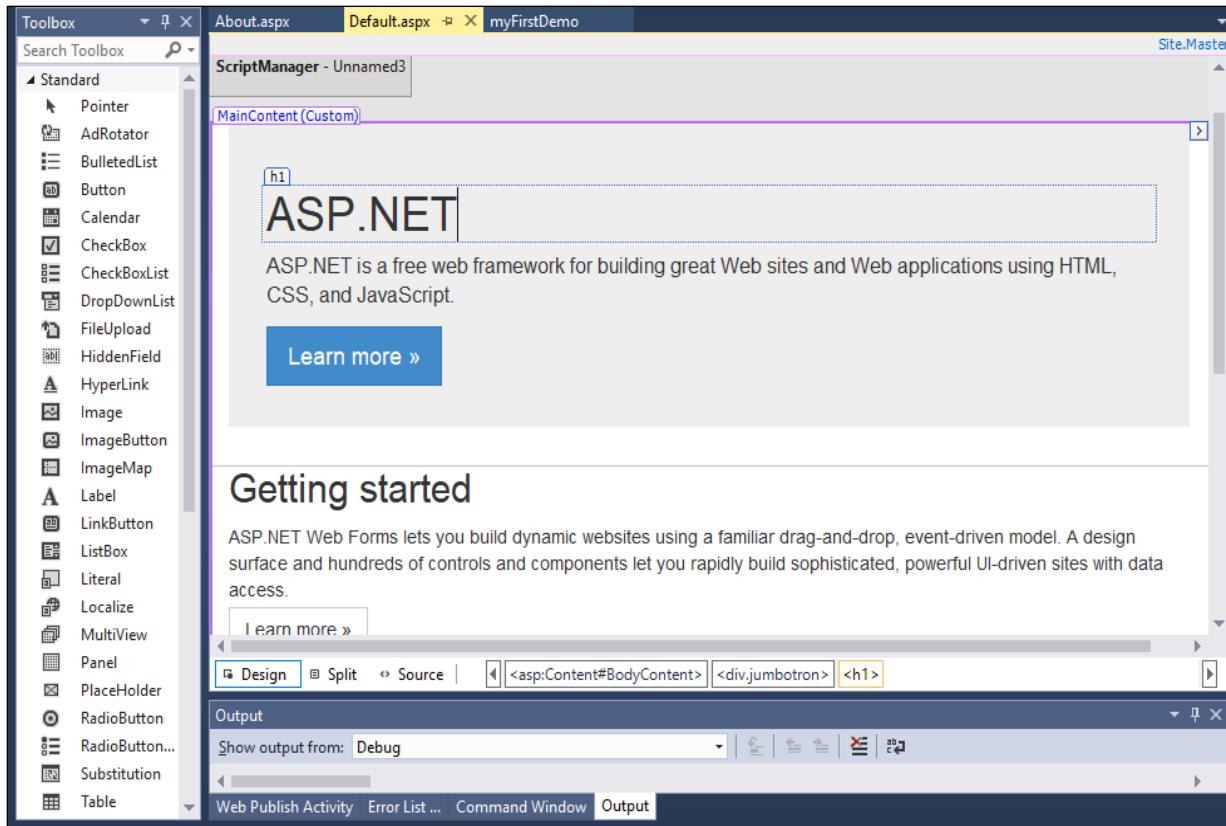
5. Click **Create** and the project is created.
6. After the project is created, there are a few default folders and pages that are added to it. Using **Solution Explorer**, these can be viewed, as shown in Figure 2.8.



**Figure 2.8: Solution Explorer**

A default page is auto-generated and can be modified or deleted if required. When the project is executed, it is the default.aspx page that is displayed.

7. Select the **Default.aspx** page from the Solution Explorer and click **Design**. A UI will be displayed and a clear picture of how the content is created is shown. Here, the content, images, and alignment of the text can be modified as per the requirement as shown in Figure 2.9.



**Figure 2.9: Design Output**

8. Next, delete the Default.aspx page so that you can add a new Web Form.
9. Right-click the selected project in **Solution Explorer**.
10. Select **Add → New Item**, as shown in Figure 2.10.

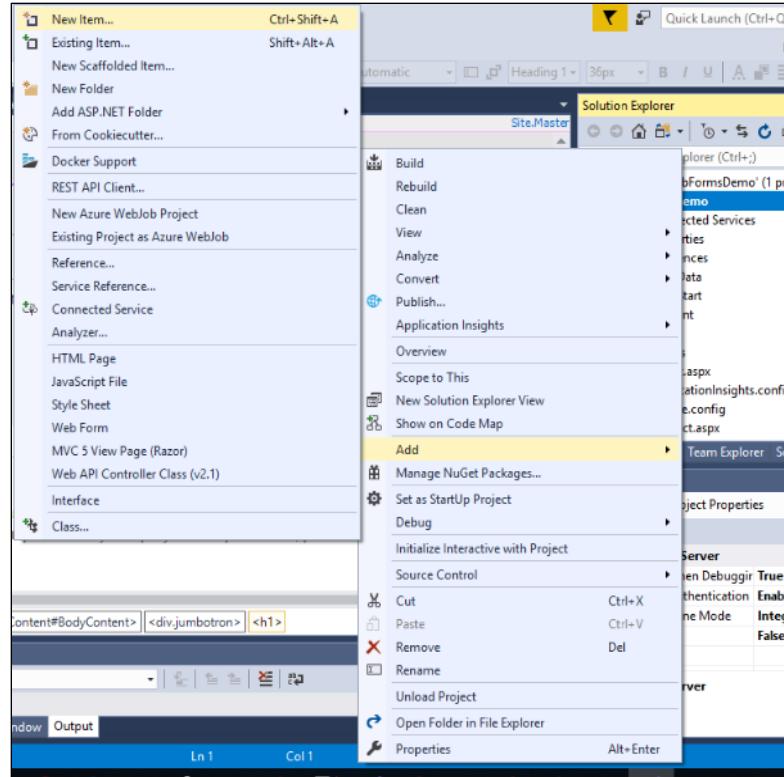


Figure 2.10: Adding a New Item

11. In the dialog box, select **Web Form** and name it as **Default** as shown in Figure 2.11.

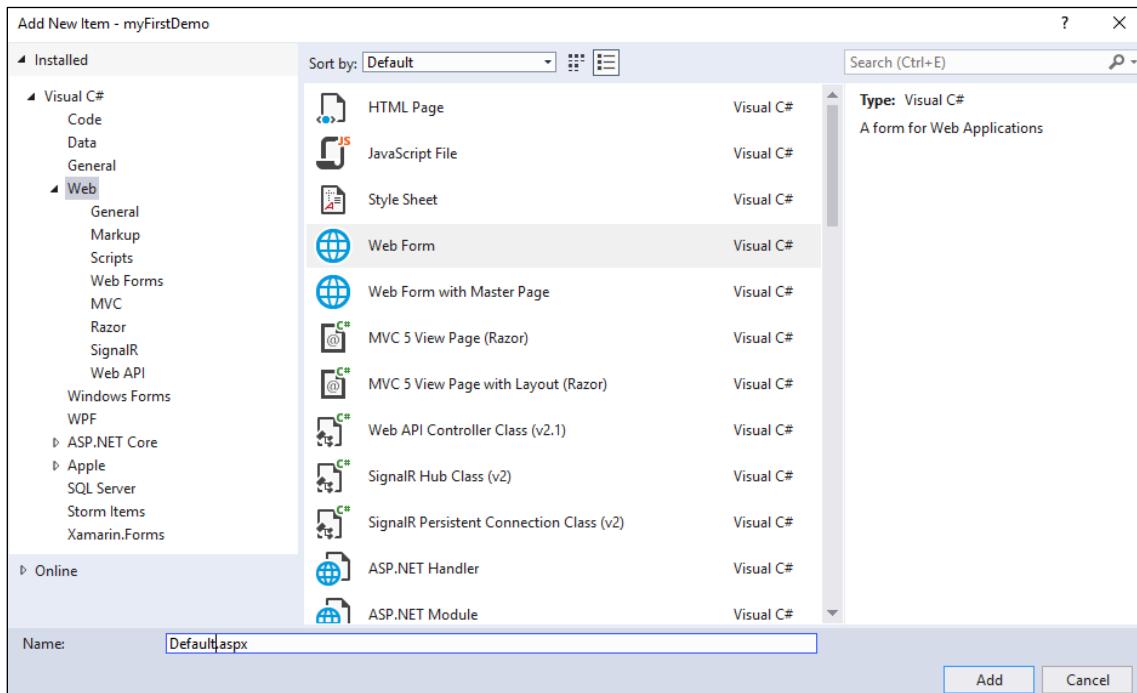


Figure 2.11: Selecting Web Form

Thus, the Default.aspx page is deleted and a new page is added containing default HTML tags. The new Default.aspx file contains the HTML and ASP code, which defines the form.

### 2.3.2 Adding Controls to the Form

The Web Form Designer window is the primary window in the Visual Studio IDE.

The Solution Explorer, Toolbox, and the Properties window are supporting windows. Designer is used for designing a form; Code editor is used for adding code to the control on the form; and the Design or source button is used for editing the Web Forms Designer from one view to the other.

On the right, in the Solution Explorer window, all default pages as well as the newly added pages can be seen.

Controls are usually visual elements that can be added to applications. In this case, controls can be added to Web Forms.

Available controls are seen in the Toolbox on the left of the editor. Web developers can easily drag and drop these controls on the Web Form for developing an application.

To add controls to the Web Form, perform following steps:

1. Switch to Design view by selecting the **Design** tab at the bottom of the screen, as shown in Figure 2.12.

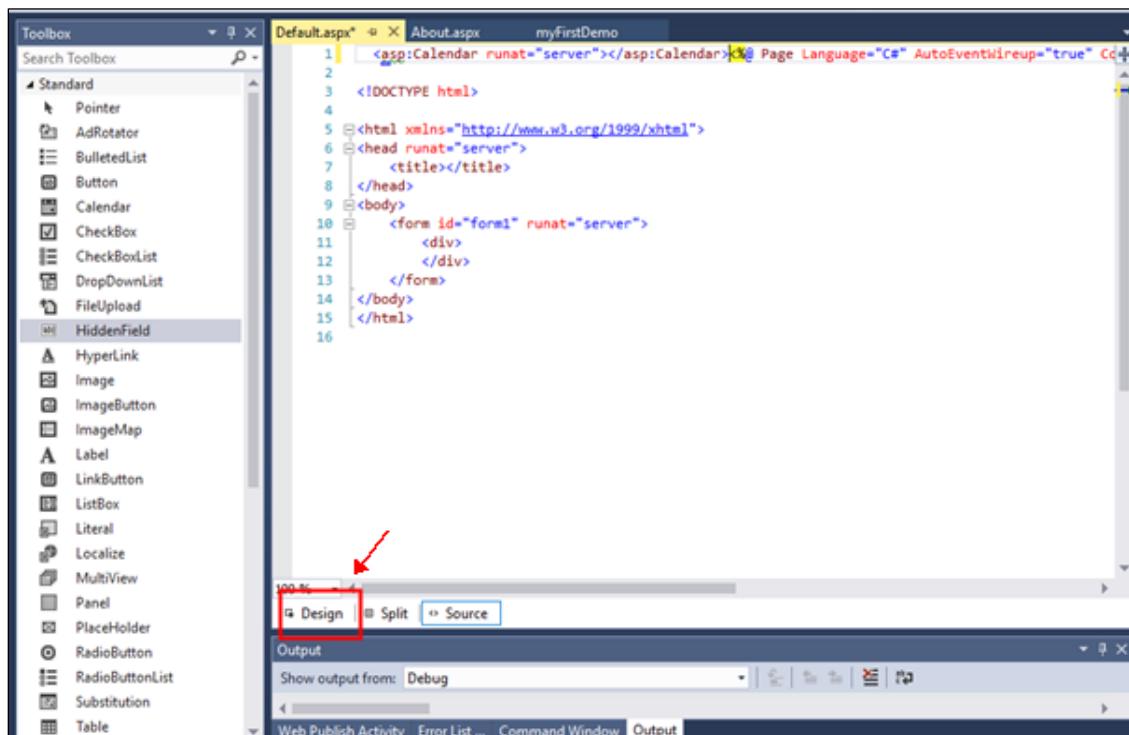
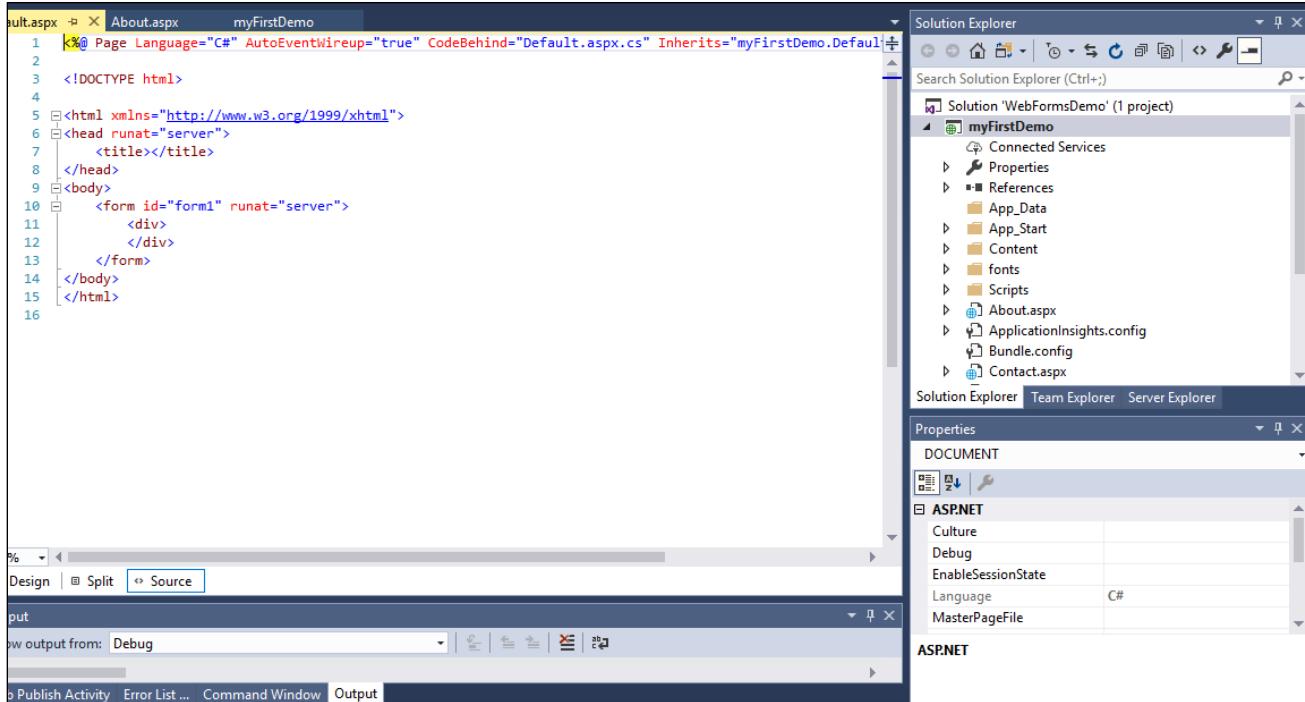


Figure 2.12: Design Tab

2. To view different controls available, select **View □ Toolbox**. Labels, text boxes, buttons, and placeholders can be seen.
3. Add a control to the form by dragging and dropping it.

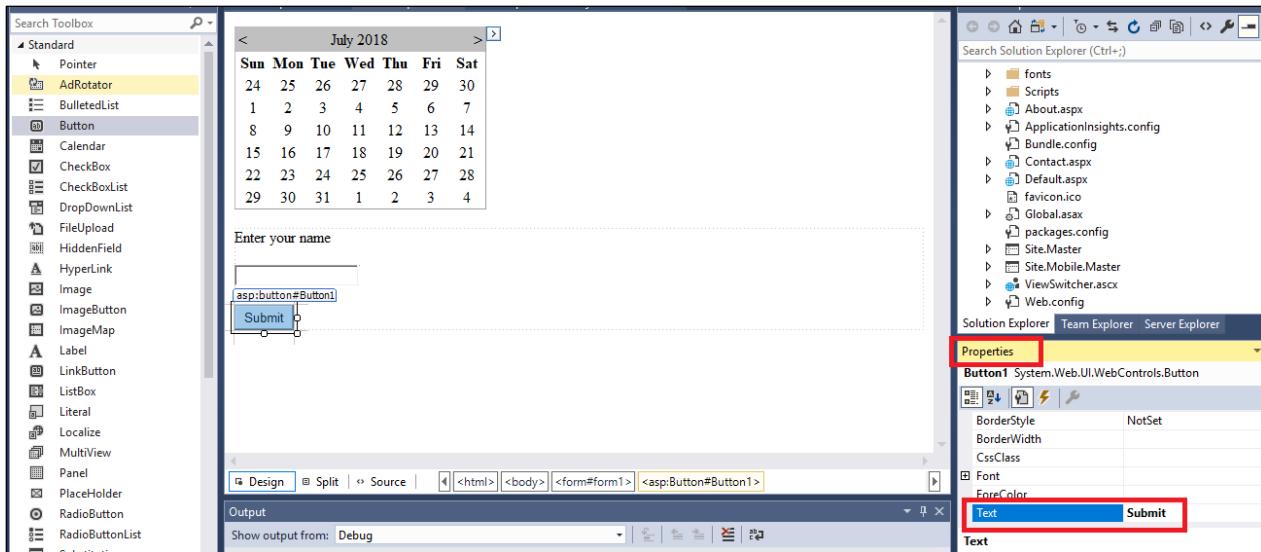
Developers can modify properties of each visual control using **Properties** window, as shown in Figure 2.13. To view the window, select **View □ Properties**.

4. Select a control to change its properties using the Properties window and specify new value for the properties.



**Figure 2.13: Selecting a Control**

5. Now, create a simple Web Form having a label, a text box, a Calendar, and a **Submit** button, as shown in Figure 2.14. Drag and drop all three controls from the toolbox into the **Design** view of the Web Form. The top **Search Toolbox** can be used to search for a required control.



**Figure 2.14: Creating a Web Form**

To design a Web application, various controls are available that can be used from Toolbox. After the required fields are generated in the designer window, the page can be viewed in the browser by hitting run or F5. In this form, when the user clicks the Submit button, after a name is entered, nothing happens. This is because no action has been specified for the button click event. The process of handling events and specifying the actions to be performed when the event occurs is called event handling.

### 2.3.3 Types of Web Forms

Table 2.1 lists different types of Web Forms.

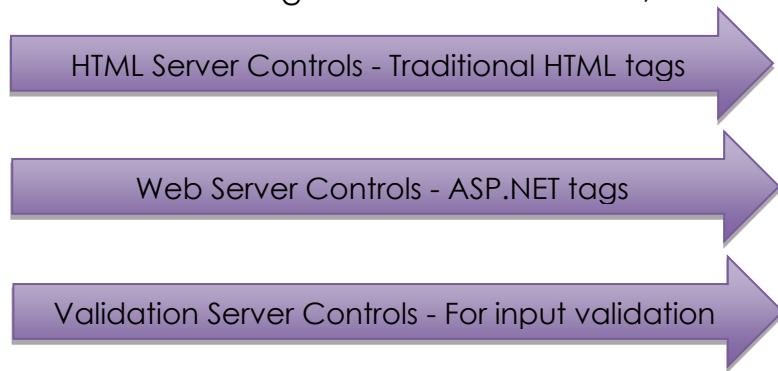
Type of Web Form	Description
Standard Web Forms	Standard Web Forms form the foundation of ASP.NET development. Developers can create dynamic Web pages by adding controls such as buttons, text boxes grids, and define event handlers to respond to user actions.
Master Pages	Master Pages offer a consistent layout and design across an entire Website. They serve as templates, defining the common structure, headers, footers, and navigation menus that are shared across multiple content pages.
User Controls	User Controls encapsulate sets of UI elements and logic, promoting code reusability. They are particularly useful for creating modular components that can be shared and integrated into various pages.
Web Parts Pages	Web Parts Pages allow users to customize the content and layout of a page. Developers can create flexible and customizable

Type of Web Form	Description
	pages by adding, removing, and personalizing Web parts according to user preferences.
Content Pages	Content Pages work in conjunction with Master Pages, focusing on the unique content for each page while inheriting the layout and structure from a Master Page. This simplifies the development of consistent layouts across a Website.
Dialog Boxes	Dialog Boxes are special Web forms used to display information or interact with users. They can be modal or modeless, serving to collect input, provide alerts, or display notifications.
Web Forms with AJAX	Integrating AJAX into Web Forms enables asynchronous communication with the server, enhancing the user experience by allowing parts of a page to be updated without requiring a full page reload.
Login and Registration Forms	They are dedicated forms for user authentication and registration processes include controls such as text boxes, buttons, and validators to collect and validate user information.

**Table 2.1: Web Forms**

## 2.4 Controls in ASP.NET

Server Controls in ASP.NET are tags that are understood and accepted by the server. There are three categories of server controls, as follows:



### 2.4.1 HTML Server Controls

Server controls permit access to various properties that use server-side scripting. The HTML server controls are Hypertext Markup Language (HTML) components containing `runat="server"` attribute. These controls facilitate server-side events and automatic state management. Advantages of HTML server controls over HTML elements are as follows:

- The HTML server controls are compiled into the assembly with the `runat="server"` attribute.
- The HTML server controls map to the corresponding HTML tags.

- The HTML server controls retain their values, whenever ASP.NET page is reloaded.
- OnserverEvent is included in most controls for commonly used events.

Properties of controls on server-side can be modified by adding the runat="server" attribute. For instance, while loading a page or during a click event, this will allow to manipulate the element's attributes. JavaScript is used to handle this on both client and server-side.

Table 2.2 lists a few HTML server controls.

Control Name	HTML Tag with Description
HtmlHead	<head>element Used to store page Title and CSS, JavaScript links.
HtmlInputButton	<input type = button submit reset> Similar to HTML Submit Button.
HtmlInputCheck	<input type = checkbox> Used to Check/Uncheck multiple options.
HtmlInputFile	<input type = file> Used to browse and upload files.
HtmlInputHidden	<input type = hidden> Used to store temporary value.
HtmlInputImage	<input type = image> Used to load and display image.
HtmlInputpassword	<input type = password> Textbox that masks password to *.
HtmlInputRadioButton	<input type = radio> Used to select single option from multiple options.
HtmlInputreset	<input type = reset> HTML Form reset button.
HtmlText	<input type = Textpassword> Similar to HTML textbox.
HtmlImage	<img> element Used to display image.
HtmlLink	<link> element Similar to HTML hyperlink.
HtmlAnchor	<a> element Similar to HTML hyperLink.
HtmlButton	<button> element Similar to HTML button.
HtmlForm	<form> element HTML form used to add input controls.
HtmlTable	<table> element HTML table to show data/control in tabular format.

**Table 2.2: HTML Server Controls**

#### 2.4.2 Web Server Controls

Web server controls integrate special controls such as menus, calendar, and a tree view control. These controls include more built-in elements, as compared to HTML server controls. Web server controls also include some complex controls such as tables and commonly used controls such as choosing dates, displaying menus, and displaying in a grid.

Web server controls provide commonly used elements from elements of HTML server controls. Web server controls have the potential to provide type-safe programming. It also detects the browser capabilities automatically and render matching markup and hence, supports themes and enables a consistent view for controls all over the Website.

Table 2.3 lists a few Web server controls.

<b>Web Server Control</b>	<b>Description</b>
AdRotator	Displays a sequence of images.
Button	Displays a push button.
Calendar	Displays a calendar.
CheckBox	Displays a check box.
CheckBoxList	Creates a multi-selection check box group.
DataGrid	Displays fields of a data source in a grid.
DataList	Displays items from a data source by using templates.
Image	Displays an image.
ImageButton	Displays a clickable image.
Label	Displays static text that displays information as a response to an action or description of how a control will behave when clicked.
ListBox	Creates a single or multi-selection drop-down list.
Table	Creates a table.

**Table 2.3: Web Server Controls**

#### **2.4.3 ASP.NET Validation Server Controls**

While creating a Web Form, end users must fill all the required fields such as login name, password, and data. A developer must verify that the fields are filled and that they have correct data. This method of checking whether the values entered are correct is known as Validation and is used to validate the data of an input control. It will throw an error message to the user, if the data does not pass validation.

ASP.NET validation controls can be utilized when there is a requirement to create the form specifying the required ASP.NET validation controls.

A simple drag and drop of the ASP.NET Validation control and writing a simple code in a Web Form describes its functionality. It minimizes the load on the server as well as reduces the developer's effort on writing JavaScript for various types of validation.

Table 2.4 lists a few various validation server controls.

Validation Server Control	Description
CompareValidator	Compares the value of one input control to the value of another input control or to a fixed value.
CustomValidator	Allows writing a method to handle the validation of the value entered.
RangeValidator	Checks that the user enters a value that falls between two values.
RegularExpressionValidator	Ensures that the value of an input control matches a specified pattern.
RequiredFieldValidator	Makes an input control a required field.
ValidationSummary	Displays a report of all validation errors that occurred in a Web page.

**Table 2.4: Validation Server Controls**

The syntax for creating a validation server control is as follows:

#### Syntax

```
<asp:control_name id="some_id" runat="server" />
```

## 2.5 Event Handling in ASP.NET

An event is generated for any action such as a key press or mouse button click. It also includes the notifications generated by the system. A process flows through events. For instance, interrupts are known as system-generated events. At the point when events occur, the application must have the capacity to react to it and handle it.

Events are handled at the server end when raised at the client end in ASP.NET. For instance, for a button click in the browser by the user, the event raised is a **Click** event. After handling the event, the browser publishes this client-side event to the server. After the event is raised, the server describes what to try next. The event handler is executed, when the event message is sent to the server. It also checks if Click event has a related event handler or not.

ASP.NET event handlers use two parameters and return `void`.

#### Syntax

```
private void Event Name (object sender, EventArgs e);
```

### 2.5.1 Application and Session Events

Following are the most significantly used application events:

**Application\_Start** – This event is raised when the application/Website is started.

**Application\_End** – This event is raised when the application/Website is closed.

Following are the most commonly used session events:

**Session\_Start** - This event is raised when a user requests a page from the application.

**Session\_End** - This event is raised when the session ends.

### 2.5.2 Application and Session States

Application state in ASP.NET indicates data storage, which is available for all the classes in the application. The state has quick access over the data/information stored in the database. The state can be applied for all the sessions and users.

The instance `HttpApplicationState` is created when a user accesses resources for the first time. The state is always saved in the memory of a server. Thus, it is lost when the application is closed or restarted by the user.

Session state enables a user to store and access values as the user browses the pages in an application. In a short span of time, the state checks for the requests from the browser. The session state is always enabled for all the variables that are stored in the `SessionStateItemCollection` object.

### 2.5.3 Page and Control Events

ASP.NET has various page and control events. The events are explained in sequence of their working, as follows:

1. **PreInit:** This event is raised to create or re-create dynamic controls. Check the `IsPostBack` property to determine whether this is the first time the page is being processed. The `IsCallback` and `IsCrossPagePostBack` properties have also been set at this time.
2. **Init:** This event is raised after all controls have been initialized.
3. **InitComplete:** This event is raised at the end of the page's initialization stage.

4. **PreLoad:** This event is utilized to execute processing on the page or control before the processing of load event.
5. **Load:** This event is raised when the page or a control is loaded.
6. **PreRender:** This event is raised when the page or the control is to be rendered.
7. **LoadComplete:** This event is utilized when the controls are loaded on the page.
8. **Render:** This event writes out control markup to send to the Web browser.
9. **Unload:** This event is raised when the page or control is unloaded from memory.

#### **2.5.4 Event Handling for Controls**

ASP.NET controls are applied as classes and they contain events that are triggered when a user executes an action on them.

For instance, when a user clicks a button, the event 'Click' is generated. In-built attributes and event handlers are used for handling events. The Visual Studio application, by default, creates an event handler by including the `Handles` clause and a Sub procedure.

#### **2.5.5 Creating Event Handlers in Visual Studio**

To create a default event handler, double-click the control in Design view for which event handler must be created.

Visual Studio creates a handler for the default event and then, the code editor is opened in the event handler along with the insertion point.

The code for creating a simple ASP.NET button control is shown in Code Snippet 1.

#### **Code Snippet 1**

```
<asp:Button ID="btnCancel" runat="server" Text="Cancel"
/>
```

Table 2.5 lists some of the common control events.

Event	Attribute	Control
Click	OnClick	Button, ImageButton, LinkButton, and ImageMap
Command	OnCommand	Button, ImageButton, and LinkButton
TextChanged	OnTextChanged	TextBox

Event	Attribute	Control
SelectedIndexChanged	OnSelectedIndexChanged	DropDownList, ListBox, RadioButtonList, and CheckBoxList
CheckedChanged	OnCheckedChanged	CheckBox and RadioButton

**Table 2.5: Common Control Events**

Assuming that a button control with ID btnSubmit and text Submit has been created in the Web Forms application, when a developer double-clicks the control in Design, following event handler will be auto-generated:

```
protected void btnSubmit_Click(object sender, EventArgs e)  {  
}
```

Developer can then add appropriate code as per requirement in the event handler. For example, Code Snippet 2 shows an example of displaying a custom message through a Label control.

#### Code Snippet 2

```
protected void btnSubmit_Click(object sender, EventArgs e)  {  
    lblMessage.Text = "Hi, Greetings and Good day.";  
}
```

## 2.6 Exception Handling in ASP.NET

Exceptions are run-time errors that disrupt the execution flow of instructions in a program. Upon encountering an exception, the application terminates without permitting the code to continue executing.

In ASP.NET, developers can handle these exceptions by using the try-catch or try-catch-finally constructs. The try block contains statements that can possibly cause exceptions; a catch block is designed to handle exceptions.

The finally block, which executes regardless of whether an exception occurred or not, can be used to handle the clean-up process.

### Syntax

```
try{  
    // Statements that may cause exception  
}  
catch (ExceptionClassName x){  
    // Statements to handle exception  
}  
finally{  
    // Statement to clean up  
}
```

Code Snippet 3 shows an example using try-catch-finally block in ASP.NET.

### Code Snippet 3

```
<script runat=server>
public Page_Load(sender As Object, e As EventArgs) {
try{
    // Statements that may cause exception
}
catch (ExceptionClassName x){
    // Statements to handle exception
}
finally{
    // Statement to clean up
}
}
</script>
```

#### 2.6.1 Unhandled Exceptions

When an exception occurs unexpectedly and is out of scope of the try/catch/finally block, the page error and the application error event handlers are used. Thus, when there is no way to handle an exception that has occurred abnormally, it is handled at either the page level or the application level. If handled at Page level, details should be provided for the `Page_Error` event.

#### Syntax

```
public void Page_Error(Object sender, EventArgs
e)
{
    // Implementation here
}
```

The `Application_Error` event can also be used to handle the same exception.

```
public void Application_Error(Object sender,
EventArgs e)
{
    // Implementation here
}
```

#### 2.6.2 Working with Custom Errors

Code Snippet 4 depicts the default setting available in `machine.config` file of an ASP.NET Web application.

#### Code Snippet 4

```
<customErrors mode="RemoteOnly"/>
```

The `mode` attribute contains three options that can be used for customizing the error, which are as follows:

RemoteOnly	On	Off
If a custom error page exists, it is displayed to the remote user when any exception occurs.	The detailed ASP.NET error page is not shown to local users as well. This page is displayed when a custom error page is available.	The detailed ASP.NET error page is displayed always, even if a custom error page exists.

In addition to the mode settings, several other configuration options are available for the `customErrors` section of the configuration. One of the most important ones is `defaultRedirect`. When an error occurs, the client is redirected to a custom error page, as shown in Code Snippet 5.

#### Code Snippet 5

```
<customErrors mode="On" defaultRedirect=" errorPage.aspx"/>
```

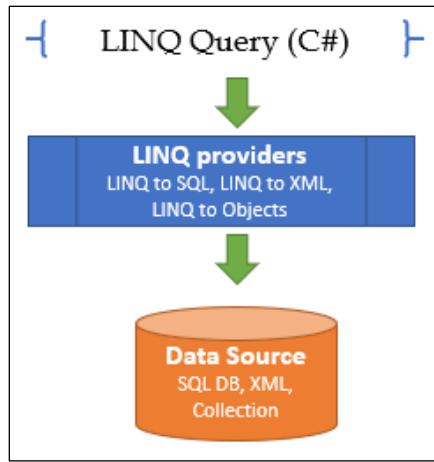
## 2.7 Working with LINQ

Language Integrated Query (LINQ) is a C# query framework that allows you to access data from a variety of sources and formats. It enables to process information, whether it is data from or to a database, XML, or dynamic data. It also provides a single query interface for several data sources. LINQ also has simple syntax.

It is used to facilitate consistent access to numerous data sources such as databases and XML using the same syntax as to access different data sources from within the language itself. LINQ benefits from strong typing features such as compile time checking for errors. The three main LINQ data providers by Microsoft are as follows:



Figure 2.15 displays LINQ providers as a bridge between LINQ and the data sources such as SQL and XML.



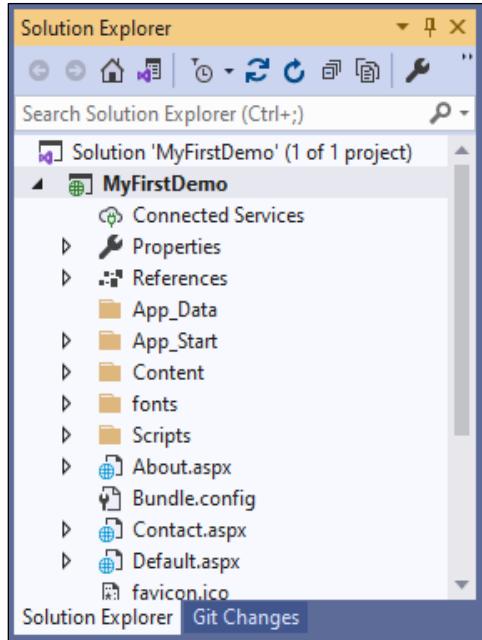
**Figure 2.15: LINQ Providers**

LINQ supports deferred execution, which evaluates a LINQ expression until its realized value is required. It greatly improves performance by avoiding unnecessary execution. Deferred execution is applicable on any data such as LINQ-to-SQL, LINQ-to-Entities, LINQ-to-XML, or in-memory collection.

### 2.7.1 Creating an Application with LINQ from XML

Launch Visual Studio 2022. Then, perform following steps:

1. Open the **MyFirstDemo** solution that was created earlier, as shown in Figure 2.16.



**Figure 2.16: MyFirstDemo Solution**

2. Next, add **XML File** under **Data** to the same application and name it as Employee as shown in Figure 2.17.

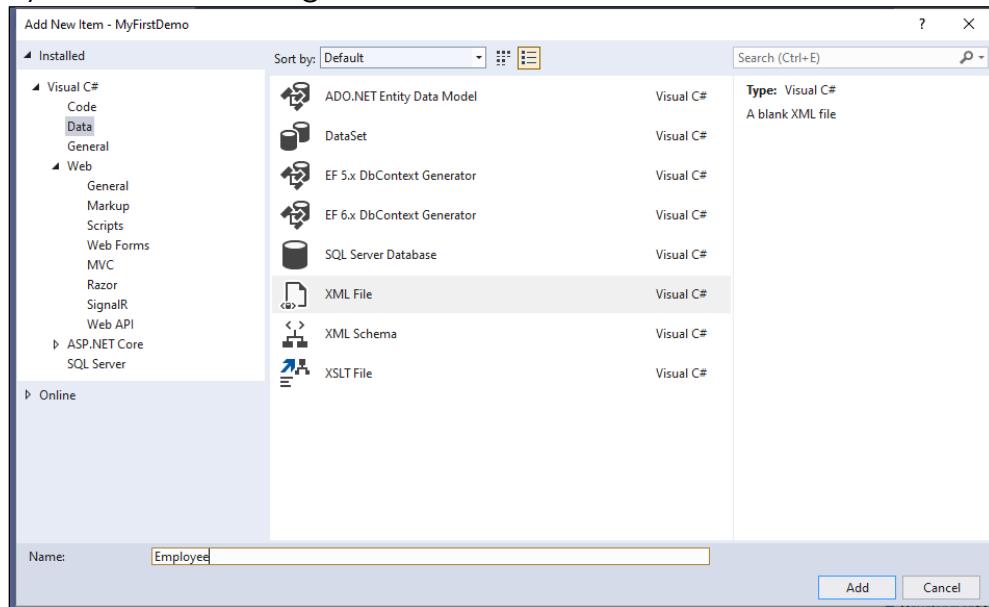


Figure 2.17: Naming the XML File

3. Next, create Employee data as shown in Figure 2.18.

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
  <Employee>
    <EmpCode>EMP1001</EmpCode>
    <EmpName>Jose</EmpName>
    <EmpLocation>Australia</EmpLocation>
  </Employee>
  <Employee>
    <EmpCode>EMP1002</EmpCode>
    <EmpName>Rosie</EmpName>
    <EmpLocation>Poland</EmpLocation>
  </Employee>
</Employees>
```

Figure 2.18: Employee.xml

4. Open **Default.aspx** page, remove the existing controls, and add the LINQ code as shown in Figure 2.19.



The screenshot shows the Visual Studio IDE with the code editor open. The title bar says "Default.aspx\* Employee.xml MyFirstDemo: Overview". The code is written in C# and uses inline ASP.NET code blocks (like <%> and <%#>) to embed LINQ code directly into the ASPX page.

```
9<%
10<form id="form1" runat="server">
11<div>
12<%>
13    XElement xelementEmployee = XElement.Load(@"C:\Users\username\source/repos\MyFirstDemo\MyFirstDemo\Employee.xml");
14    var result = from list in xelementEmployee.Elements("Employee")
15                 where (string)list.Element("EmpLocation") == "Australia"
16                 select list;
17    foreach (XElement xEle in result)
18    {
19        <%>
20            <%# =xEle%>
21        <%>
22    }
23<%>
24</div>
25</form>
26
```

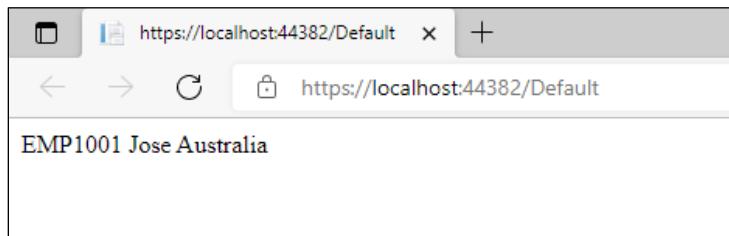
**Figure 2.19: LINQ Code**

In the example, `XElement` type is used that loads an XML file into memory and parses the XML. It allows removal of old code and eliminates the possibility of bugs and typos.

The reference of XML content, two employee records, is loaded within `xelementEmployee` object. LINQ query using `from` `where` and `select` syntax is applied. This is done to filter information for a specific country under column `EmpLocation`. Once the collection is available within `result` object, `foreach` is used to traverse through the collection element called `result`.

Use of `<% %>` within .aspx page gives the flexibility of using Code-behind kind of code within page.

The output of the code is as displayed in Figure 2.20.



**Figure 2.20: LINQ Code Output**

5. Minor changes can be done to get only name, using `xEle.Element()` to provide specific column name as shown in Figure 2.21.

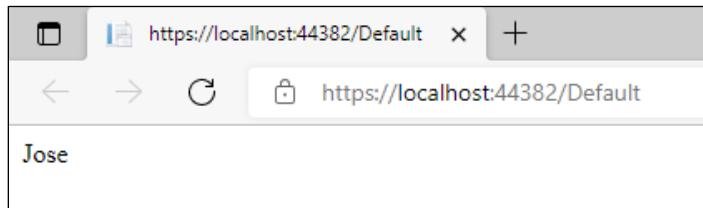
```

17     foreach (XElement xEle in result)
18     {
19         %>
20         <% =xEle.Element("EmpName")%>
21     }
22 }

```

**Figure 2.21: Using xEle.Element**

Output of this LINQ query is as shown in Figure 2.22.



**Figure 2.22: Modified Output**

## 2.8 Data Caching

One of the significant strategies of ASP.NET is caching, which helps increase the performance of an application. Caching refers to the act of storing anything in memory that is frequently used in order to improve performance. Output cache is used to increase the performance of an ASP.NET MVC application.

Output Cache filter attribute in ASP.NET MVC is similar to output caching in Web Forms. Output cache is used to save the content returned by a controller operation. It refers to the ability to save the output of a controller in memory. As a result, all requests for the same operation in that controller will return the cached result. This eliminates the requirement to generate the identical content each time the same controller action is called.

The Output Cache filter allows developers to cache data from an action method's output. This attribute filter caches data for 60 seconds by default after which ASP.NET MVC caches the result again.

Code Snippet 6 displays the code for implementing Output Cache.

### Code Snippet 6

```

[OutputCache(Duration = 20, VaryByParam = "*")]
public ActionResult Test()
{
    var i = Int32.MaxValue;
    System.Threading.Thread.Sleep(4000);
    return Content(Count++);
}

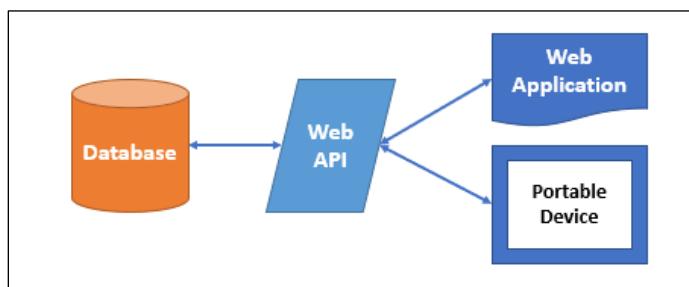
```

In the Code Snippet, the result of the `Test()` action method is cached for 20 seconds by default.

## 2.9 Web API

Application Programming Interface (API) refers to a technology for communication between applications. Web API refers to communication between Web applications. It uses HTTP protocol. HTTP features, such as Uniform Resource Identifiers (URIs), request/response headers, caching, versioning, and various content formats. No extra configuration settings for various devices must be defined using Web API.

Figure 2.23 shows the Web API framework.



**Figure 2.23: Web API Framework**

In the current trend, people use mobile, and tablets that contain apps rather than Web-based applications. To display any service data to these browsers and modern device apps, an API that is compatible with both should be used. For example, a user can use Twitter, Facebook, and Google API on Web as well as phone apps. In order to expose data and service to different devices, developers should use the Web API framework. This is an open-source platform and is ideal to build RESTful services.

REST is a set of guidelines or principles applied to the architectures available in a network system. REST is neither a protocol nor a standard; it is an architecture-style on which systems are designed consisting of protocols, data components, hyperlinks, and clients. RESTful Web services are Web services based on REST architecture and are accessed using HTTP protocol on the Web.

ASP.NET Web API is a unique technology that makes it easy to create distributed applications. It is simple to create HTTP services that work with a variety of customers including different sorts of devices and browsers. On the .NET Framework, ASP.NET Web API is an excellent foundation for developing RESTful apps.

Following are the features of Web API:

Supports convention-based Create, Read, Update, and Delete (CRUD) actions as it operates with HTTP methods such as `POST` and `DELETE`

Includes an accept header and HTTP status code to the responses

Does formatting of responses into `JSON`, `XML`, or any format by `Web MediaTypeFormatter`, to add as a `MediaTypeFormatter`

Becomes more simple and robust using routing, controllers, action results, and filter. Other than these MVC features, others include model binders, IOC container, or dependency injection

Accepts and generates content, such as images, PDF files, and so on

Supports Open Data Protocol (OData) automatically. Clients use the method for OData query composition. Here, the controller method to which a new `[Queryable]` attribute is applied returns `IQueryable`

Hosts within the application or on Internet Information Server (IIS)

## 2.10 Web Security

Implementing security requires authentication, encryption, and authorization. When a user tries to access a Website, user identification and authentication is to be implemented. Following mechanisms are used for user authentication:

### Windows Authentication

Used for Intranet applications, wherein an organization can extend the functionality of users available over intranet.

### Forms Authentication

Used for Internet applications, where different user credentials have been stored in database or other means of data storage.

### Passport Authentication

Paid Service by Microsoft that offers a single logon or Single Sign On.

Encrypting the communication between the client browser and the Web server ensures confidentiality. The process of identifying and assigning specific responsibilities to individual users is known as authorization. Integrity refers to the preservation of data's integrity through a variety of methods, such as implementing a digital signature.

# Summary

- ✓ ASP.NET Web Forms is one of the best methods to create ASP.NET Websites and Web-based applications. A Web Form can be created using Visual Studio and Web Form templates.
- ✓ ASP.NET Web Forms provide a versatile framework, including standard forms, master pages for consistent layouts, user controls for reusability, specialized forms, and an event-driven model simplifying user interaction.
- ✓ The types of server controls for input validation are namely, HTML server controls (traditional HTML tags), Web server controls (new ASP.NET tags), and validation server controls.
- ✓ Events are handled at the server end when raised at the client end in ASP.NET.
- ✓ The keywords try-catch-finally are used to implement exception handling in ASP.NET.
- ✓ Output cache is used to increase the performance of an ASP.NET MVC application.
- ✓ ASP.NET Web API is a unique technology that makes it easy to create distributed applications.

# Test Your Knowledge



1. A Web Forms page is composed of which of these two components?
4. Which Web server control displays a clickable image?

<b>A</b>	JavaScript and AJAX
<b>B</b>	HTML and ASPX
<b>C</b>	ASPX and code-behind-file
<b>D</b>	Code-behind-file and HTML

2. How does an Event occur?

<b>A</b>	A mouse click
<b>B</b>	A key press
<b>C</b>	System-generated notification
<b>D</b>	All of these

3. Which is the correct attribute for the **CheckedChanged** control?

<b>A</b>	OnClick
<b>B</b>	OnCommand
<b>C</b>	OnSelectedIndexChanged
<b>D</b>	OnCheckedChanged

<b>A</b>	Button
<b>B</b>	Image
<b>C</b>	ImageButton
<b>D</b>	Hyperlink

5. Which of the following are server controls?

<b>A</b>	HTML Server Controls
<b>B</b>	Web Server Controls
<b>C</b>	Validation Server Controls
<b>D</b>	All of these

## Answers

1	C
2	D
3	D
4	C
5	D

## **Try It Yourself**

1. Using ASP.NET and Visual Studio 2022, create a basic login page with a username and password field.
2. Using ASP.NET and Visual Studio 2022, implement an event handler for a button click. For example, when the button is clicked, display a greeting message using a Label control.
3. Implement a button click event handler using ASP.NET and Visual Studio 2022 to process the user input (for example, concatenate a greeting message).

# Session 3: Working with ADO.NET and Entity Framework

## Session Overview

This session describes the ADO.NET mechanism. It explains Entity Framework and finally, it explores database handling in ASP.NET MVC using Entity Framework.

## Objectives

In this session, students will learn to:

- ✓ Describe ADO.NET
- ✓ Explain Entity Framework
- ✓ Describe data handling in ASP.NET MVC with a code-first database

### 3.1 Overview of ADO.NET

ASP.NET applications often make use of data in some way or the other. There are various approaches that developers can use to work with data in Web applications. One of them is ADO.NET.

ADO.NET is a core component of .NET framework and is used for establishing a connection between an application and data sources. In the Microsoft.NET framework, ADO.NET provides a bridge across a wide range of database systems through a standard set of components. It can be found in the standard base class library that comes with the Microsoft.NET framework.

ADO.NET is sometimes referred to as an advancement of the ActiveX Data Objects (ADO) technology, which was immensely successful during early days of Microsoft.

### 3.2 Data Layer

Any application that is developed has an architecture and should have following layers:

Presentation Layer	Business Layer	Data Layer
The layer in which the user interface is created. In other words, it is the area that is directly visible or accessible to the client.	The layer in which validations/restrictions are defined.	The layer in which the data is stored. In simple terms, it is the storage space of the application.

Microsoft provided a solution for the data layer. The flow of development of data layers is shown in Figure 3.1. It started with Data Access Objects (DAO) and now has Entity Framework (EF).



**Figure 3.1: Microsoft Data Layers**

### **3.2.1 Data Access Objects (DAO)**

In the early days of .NET, notably in Visual Basic (VB), DAO was very popular as its object-oriented interface mapped to the Microsoft database engine. VB developers might use Open Database Connectivity (ODBC) to connect directly to Access tables and other databases. It is best used for standalone machines.

### **3.2.2 Remote Data Objects (RDO)**

RDO is an object-oriented data access interface that combines the simple capabilities of DAO with the low power and flexibility of ODBC. It was a tiny but important step toward a system that allows splitting data into objects, characteristics, and methods. It made complex parts of recorded operations and complex result sets easier to obtain.

### **3.2.3 ActiveX Data Objects (ADO)**

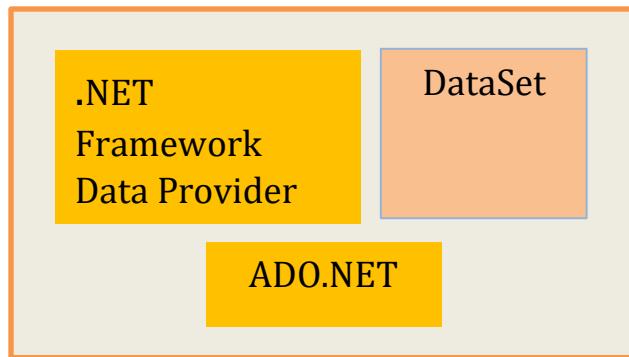
Although ADO appears to be the successor of DAO/RD, it is actually an application programming interface that allows programmers to create Windows programs that connect to a number of databases. ADO is a layer that allows application code to access any data that is stored in a generic manner without the requirement for database implementation.

This makes it easy for someone who is not familiar with databases to do complex database interactions. ADO is a package that integrates almost all of the functionalities of DAO and RDO into one. ADO is an advanced version of RDO.

### **3.2.4 ADO .NET**

Since XML was becoming popular and was a new emerging technology, therefore ADO.NET is fully based on XML. It enables consistent access to data sources such as Microsoft SQL Server, Oracle, Sybase, MySQL, and OLE DB and XML data sources. ADO.NET is a set of classes placed under assembly called `System.Data.dll`.

It comes with a large number of components that can be used to build distributed enterprise applications. There are two components of ADO.NET that can be used to access and manipulate data, .NET framework data providers, and dataset as shown in Figure 3.2.



**Figure 3.2: ADO.NET Components**

In ADO.NET, following components enable developers to read and write data from data sources:

#### Connection

This object is responsible for providing connectivity to a data source. The `SqlConnection` class in ADO.NET libraries is an example of a class that represents a connection.

#### Command

This object provides access to database commands. These are used to retrieve or modify data or execute SQL statements. For example, `SqlCommand` class has a method `ExecuteScalar()` is used to return a single value from the database.

#### DataAdapter

This object acts as a bridge between the `DataSet` object and the data source. The `DataAdapter` uses `Command` objects to execute SQL commands at the data source to both loads the `DataSet` with data and reconcile changes made to the data in the `DataSet` back to the data source. For example, `SqlDataAdapter` class contains a method called `Fill` that helps to populate the `DataTable` class object.

#### DataReader

The `DataReader` provides a high-performance stream of data from the data source. For example, `SqlDataReader` class is used to hold the data that is retuned by `ExecuteReader()` method of `SqlCommand` class. `SqlDataReader` is used to hold data.

### 3.3 Entity Framework (EF)

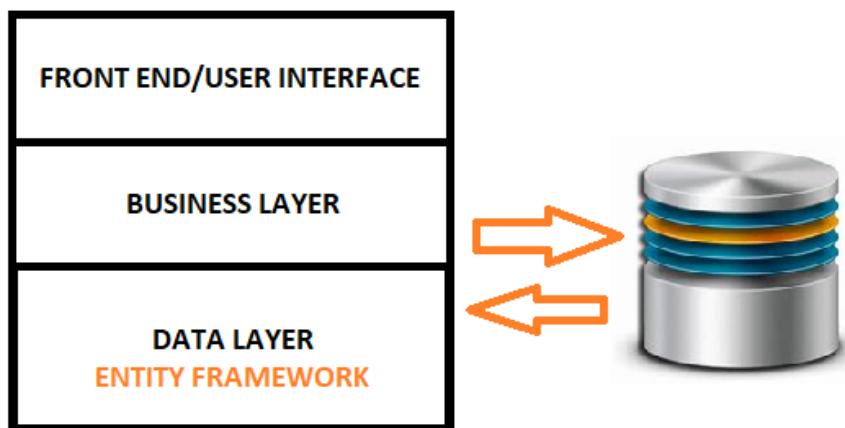
Entity Framework is an Object-Relational Mapping (ORM)-based database management framework. It is an improved version of ADO.NET that provides programmers with a completely automated database interface.

Prior to .NET 3.5 version, developers used to write ADO.NET code to store or retrieve application data from the underlying database. They used a traditional approach in the code that required initiating a connection and then, creating a result set or data set to receive or save data to the database. The next step was to transform data from the data set to .NET objects and apply business rules as required by the enterprise application. This was a time-consuming approach.

Entity Framework (EF) was first included in Visual Studio 2008 SP1 and .NET Framework 3.5 SP1. Since then, it has come a long way. Currently, EF 6.0 and EF Core are in use.

In EF, Language Integrated Query (LINQ) is utilized to access the database and interact with auto-generated code. The main benefit of auto-generated code is that it reduces overall development time for various layers, such as Model layer and Data Access layer.

The interaction of EF with the domain class and database is shown in Figure 3.3.



**Figure 3.3: Entity Framework**

EF establishes a bridge between the business entity and the data tables. It stores data that is stored in the attributes of business entities. It also retrieves data from the database and automatically converts it to business entity objects. EF executes the required database query and then, gives the results into instances of domain objects that can be used in the application.

Table 3.1 shows different versions of Entity Framework.

<b>Version</b>	<b>Release Date</b>	<b>Description</b>
EF 1.0	11 August 2008	Included with .NET Framework 3.5 SP1 and Visual Studio 2008 SP1. Criticized for issues.
EF 4.0	12 April 2010	Released with .NET 4.0, addressed criticisms from version 1.
EF 4.1	12 April 2011	Released with Code First support.
EF 4.1 Update 1	25 July 2011	Bug fixes and new supported types.
EF 4.3.1	29 February 2012	Added support for migration.
EF 5.0	11 August 2012	Targeted .NET Framework 4.5, available for .NET 4.
EF 6.0	17 October 2013	Open source project, improvements for code-first support.
EF Core 1.0	27 June 2016	Complete rewrite, cross-platform, open-source (Apache License v2).
EF Core 2.0	14 August 2017	Released with Visual Studio 2017 15.3 and ASP.NET Core 2.0.
EF Core 3.0	23 September 2019	Released with Visual Studio 2019 16.3 and ASP.NET Core 3.0.
EF Core 3.1	3 December 2019	Formally released for production, long-term supported version at least till 3 <sup>rd</sup> December 2022.
EF Core 5.0	9 November 2020	Released for production use.
EF Core 6.0	10 November 2021	Long-term supported version at least till 12 November 2024.
EF Core 7.0	8 November 2022	Released with features such as JSON columns and bulk updates.
EF Core 8.0	14 November 2023	Added features such as Value objects using Complex Types and Primitive collections.

**Table 3.1: Versions of Entity Framework**

### **3.3.1 Entity Framework (EF) Features and Benefits**

EF is an ORM tool wherein the main focus is to reduce the development load and increase productivity by reducing the unnecessary repetitive tasks while interacting with data used in the applications.

Features of EF are as follows:

- EF uses LINQ queries instead of SQL queries, which is then translated to database-specific query. It handles procedural and parameterized queries.
- EF enables caching to allow queries to be answered from the cache in case of repeat queries.
- EF allows concurrency and ensures that any changes that are being overridden are retrieved by another user.
- EF builds an Entity Data Model (EDM). This model is based on Plain Old CLR Object (POCO) and is used to query and save data. POCO is a class that is independent of any framework. POCO is the business object and has the business logic of the application.
- EF does automatic transaction management while requesting or saving data.
- EF generates the required database commands for Create, Read, Update, and Delete (CRUD) operations and then, executes them.

### **3.4 EF and EF Core Workflows**

EF and EF Core have a layer that maps between domain classes and schema. It can be implemented as one of following three approaches:

- o Database-first Approach
- o Code-first Approach
- o Model-first Approach

#### **➤ Database-first Approach**

In the database-first approach, the first step is to create a database with all the tables and procedures. EDM automatically creates domain classes. This approach focuses on the design of the database.

##### **Advantages**

- Already existing databases can be formalized for development.
- Development process is faster because classes are auto generated.
- Database schema is well defined.

#### **➤ Code-first Approach**

In the code-first approach, domain classes are created first and then, EF wizard generates database tables based on the code.

##### **Advantages**

- Database sync is easier for environments using EF migration.
- Developer has full flexibility and control over the code.

#### **➤ Model-first approach**

In the model-first approach, the developer designs entities and related elements on the EDMX platform. Next, through the EF wizard, after the selection of the

designer model and entity creation, domain classes and databases are auto-generated.

This approach is beneficial in the case of enterprise architecture.

### **Advantages**

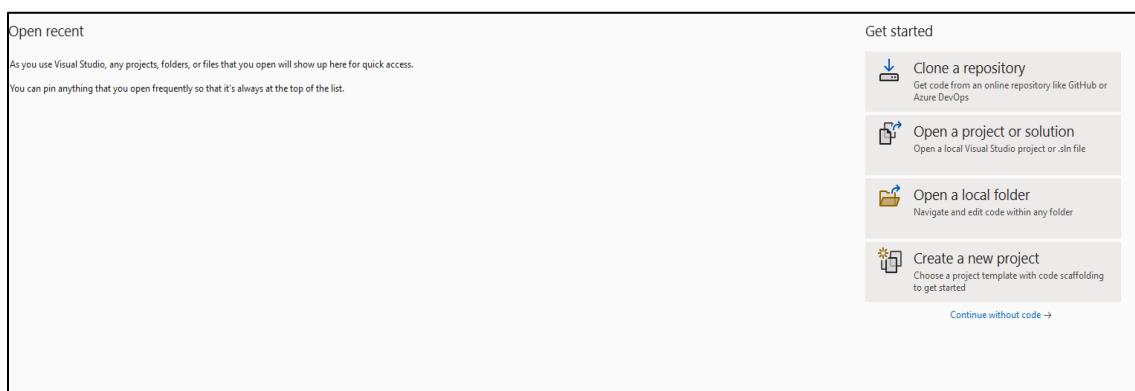
- UML designer-friendly approach.
- Very minimum programming exposure is required.

However, in the model-first approach, the structure is handled automatically so there is minimal control over the database and code.

## **3.5 Data Handling in ASP.NET MVC with a Code-first Database**

To create an application through a code-first approach in ASP.NET Core MVC, following steps are performed:

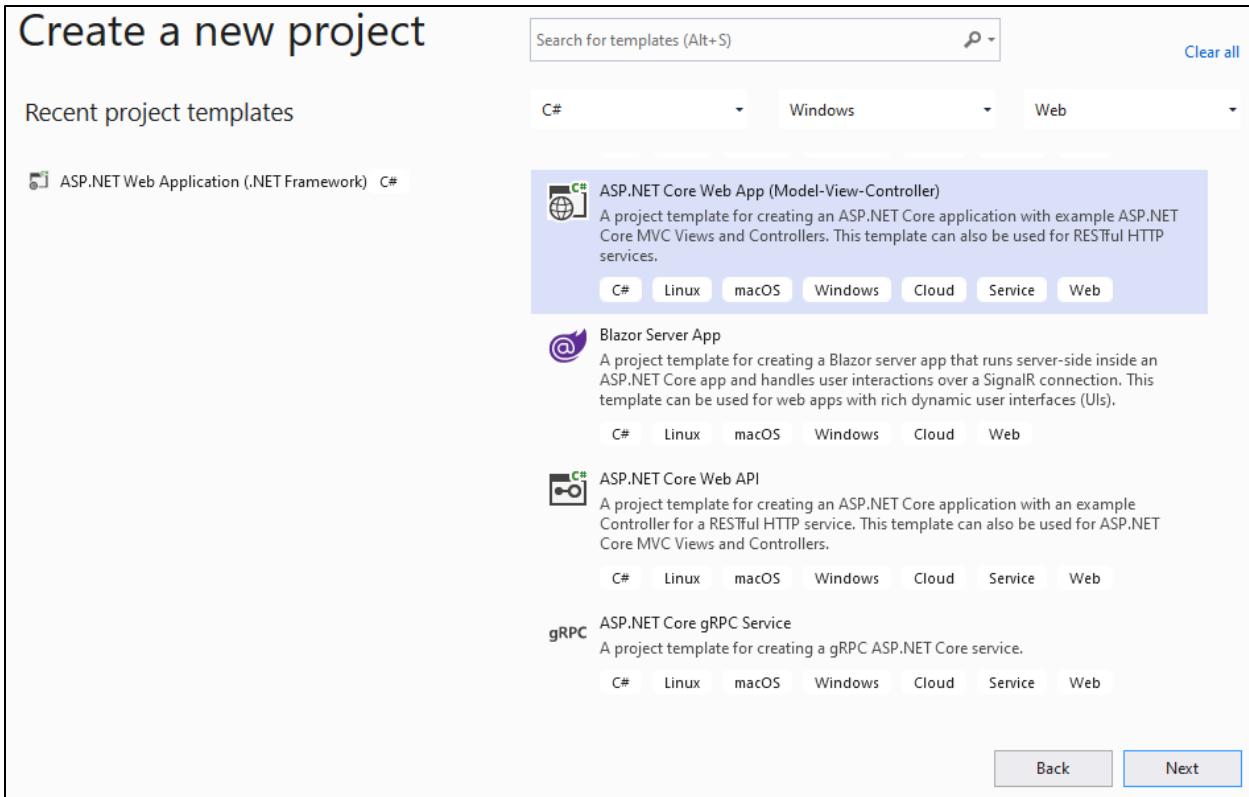
1. Open Visual Studio 2022. To create a new project, select the **Create a new project** option as shown in Figure 3.4.



**Figure 3.4: Create New Project Using Visual Studio 2022**

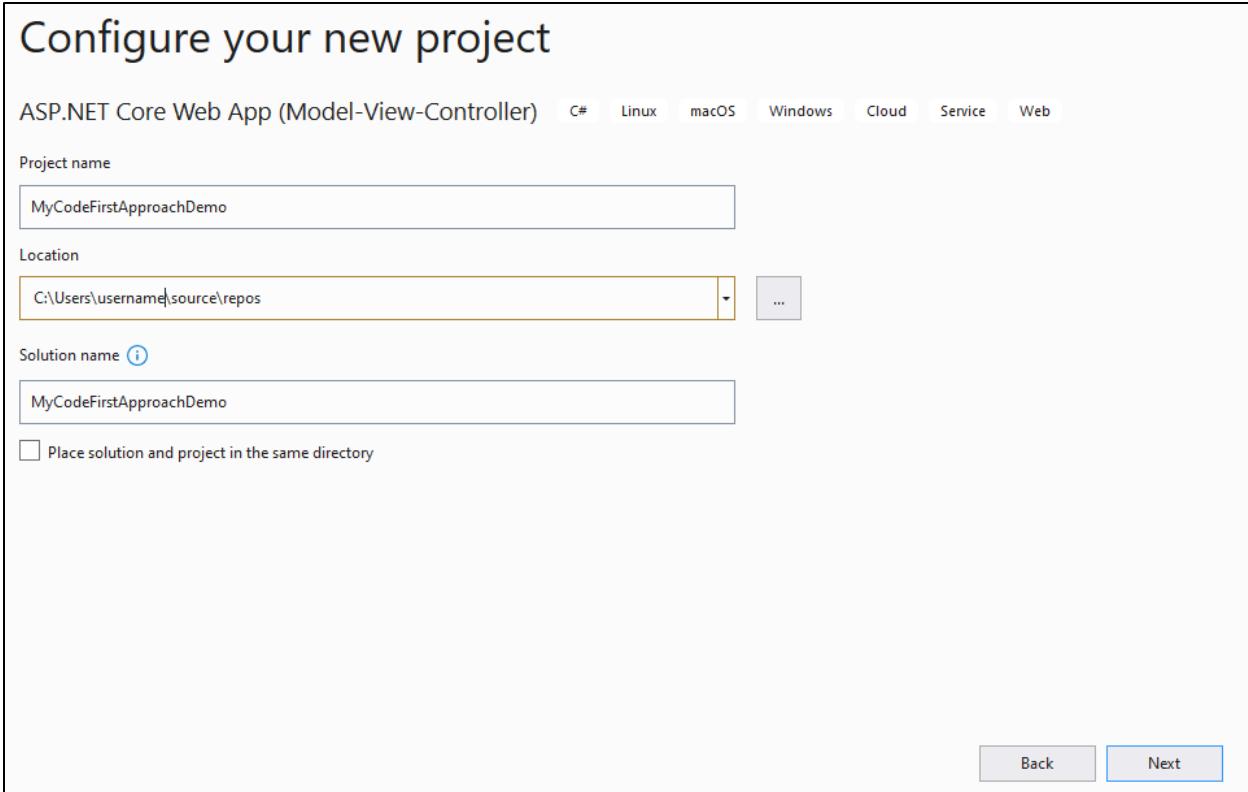
**Create a new project** window is displayed.

2. To create a Web app, select the **ASP.NET Core Web App (Model-View-Controller)** option and then, click **Next** as shown in Figure 3.5.



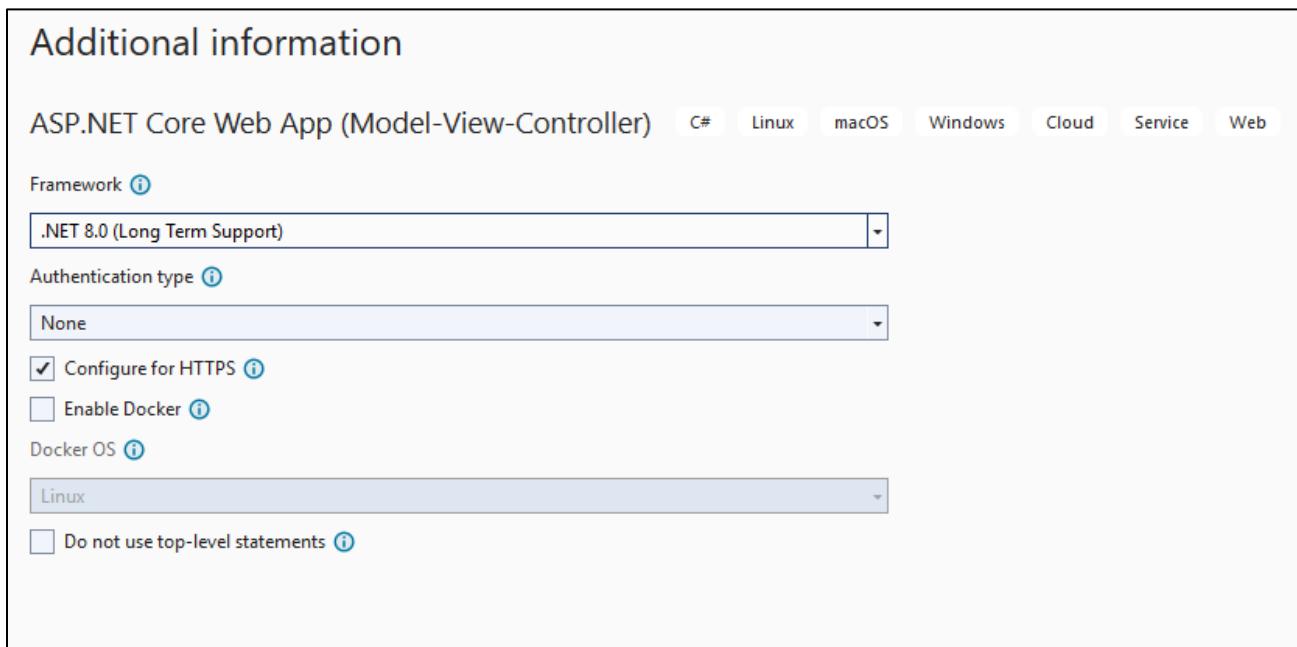
**Figure 3.5: ASP.NET Core Web App**

3. In the **Configure your new project** window, add the project name **MyCodeFirstApproachDemo** and select the location to save the project as shown in Figure 3.6.



**Figure 3.6: Configure Your New Project**

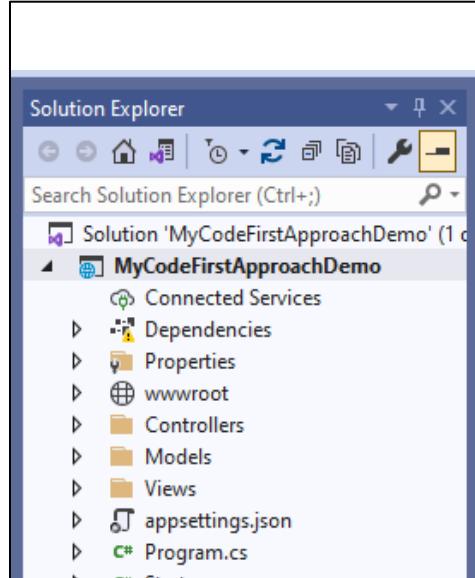
4. In the **Additional information** window, select **.NET 8.0 (Long Term Support)** and click **Create** as shown in Figure 3.7.



**Figure 3.7: Specifying Additional Information**

In the **Solution Explorer**, note the files and folders that are created.

5. Select **Dependencies**. Refer to Figure 3.8.

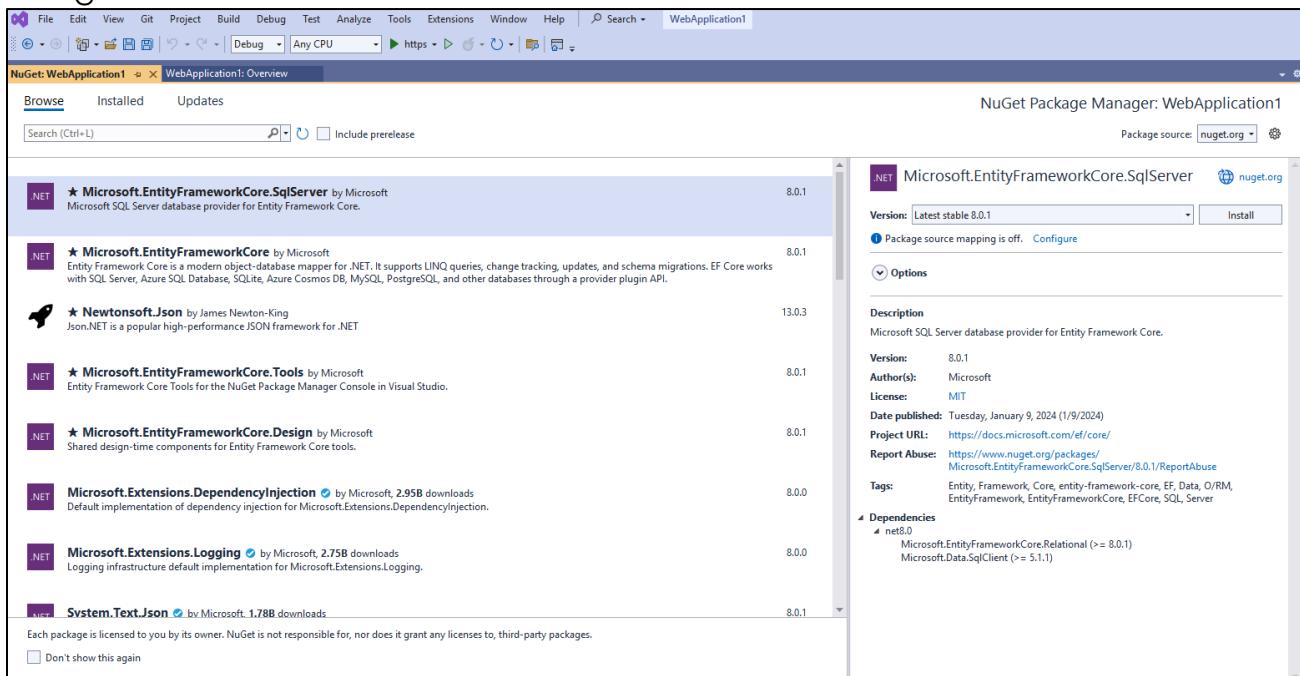


**Figure 3.8: Solution Explorer Showing Dependencies Node**

6. Right-click **Dependencies** and select **Manage NuGet Packages**.

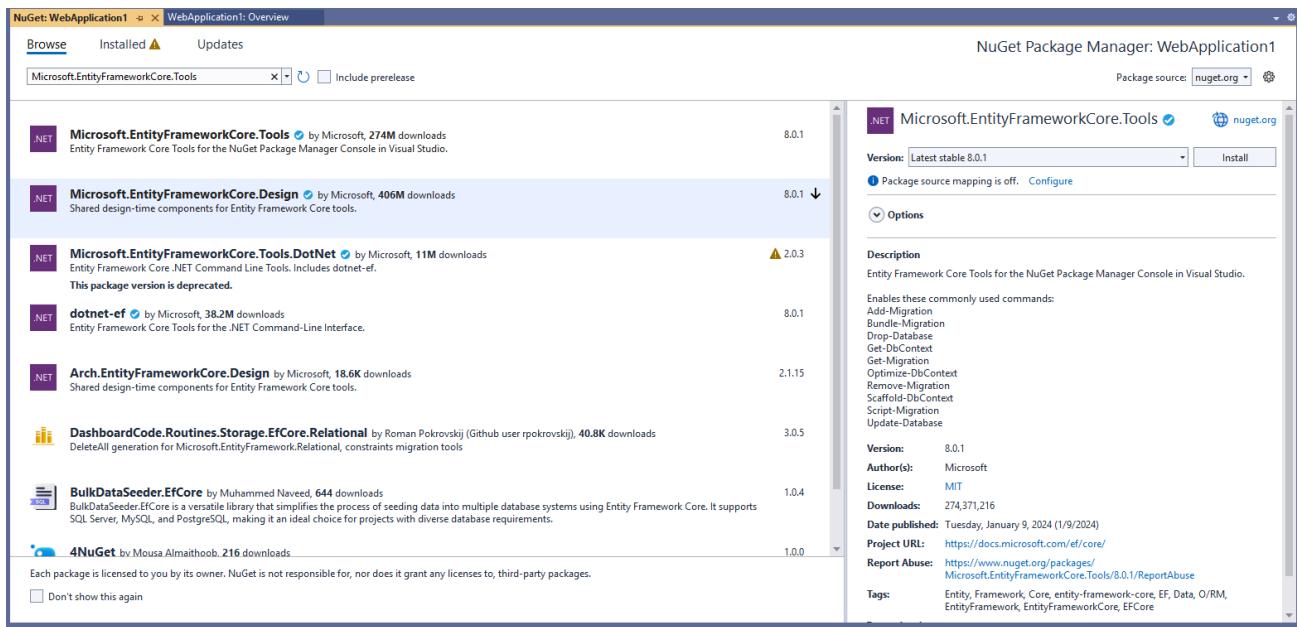
NuGet Manager is displayed.

7. To install EF SQL Server provider, search for **Microsoft.EntityFrameworkCore.SqlServer** and then, click **Install** as shown in Figure 3.9.



**Figure 3.9: NuGet Package Manager**

- Accept the license agreement. The provider will be installed.
- Next, search for **Microsoft.EntityFrameworkCore.Tools** and then, click **Install** as shown in Figure 3.10.

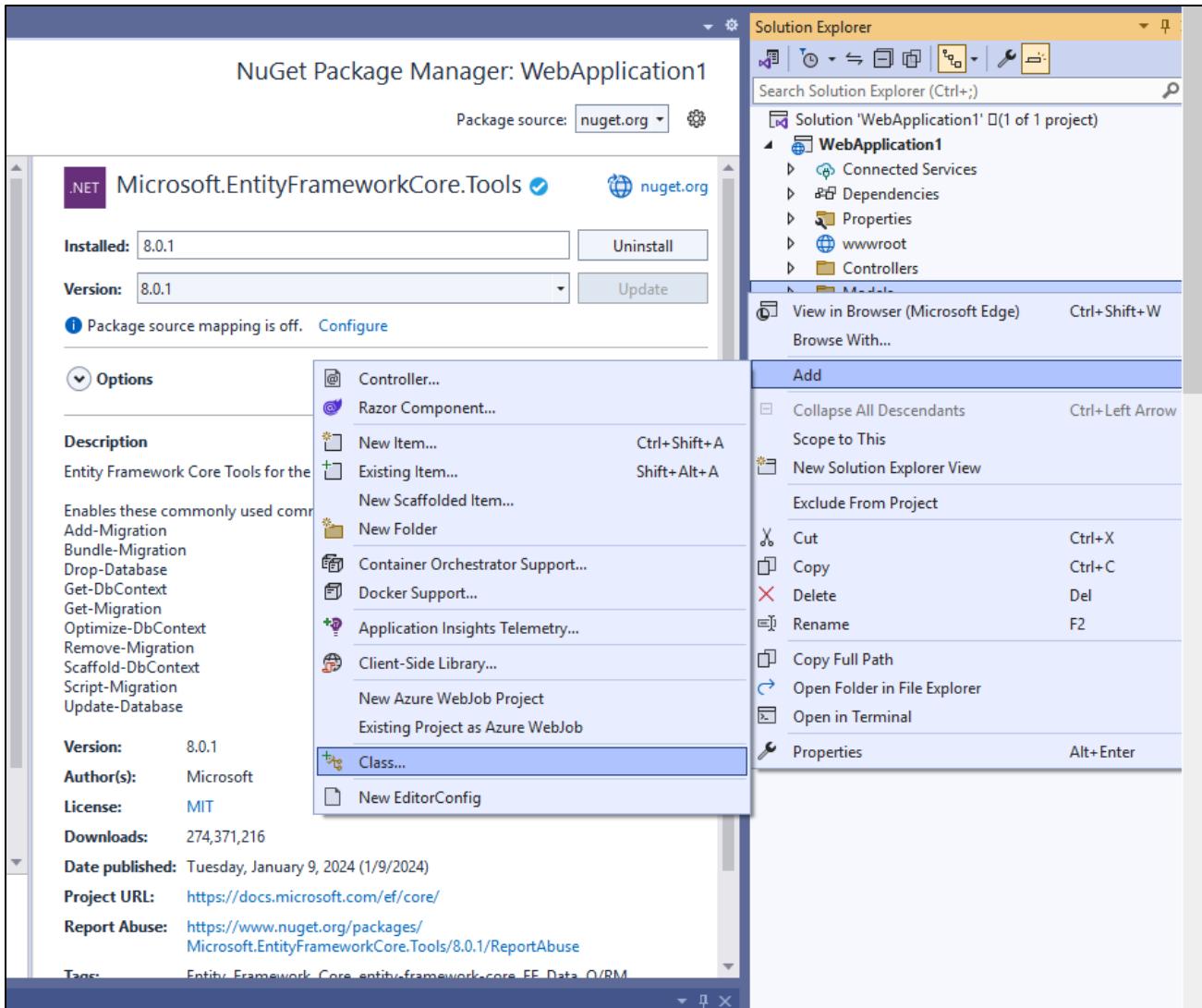


**Figure 3.10: EF Tools**

- Accept the license agreement. The Tools will be installed.

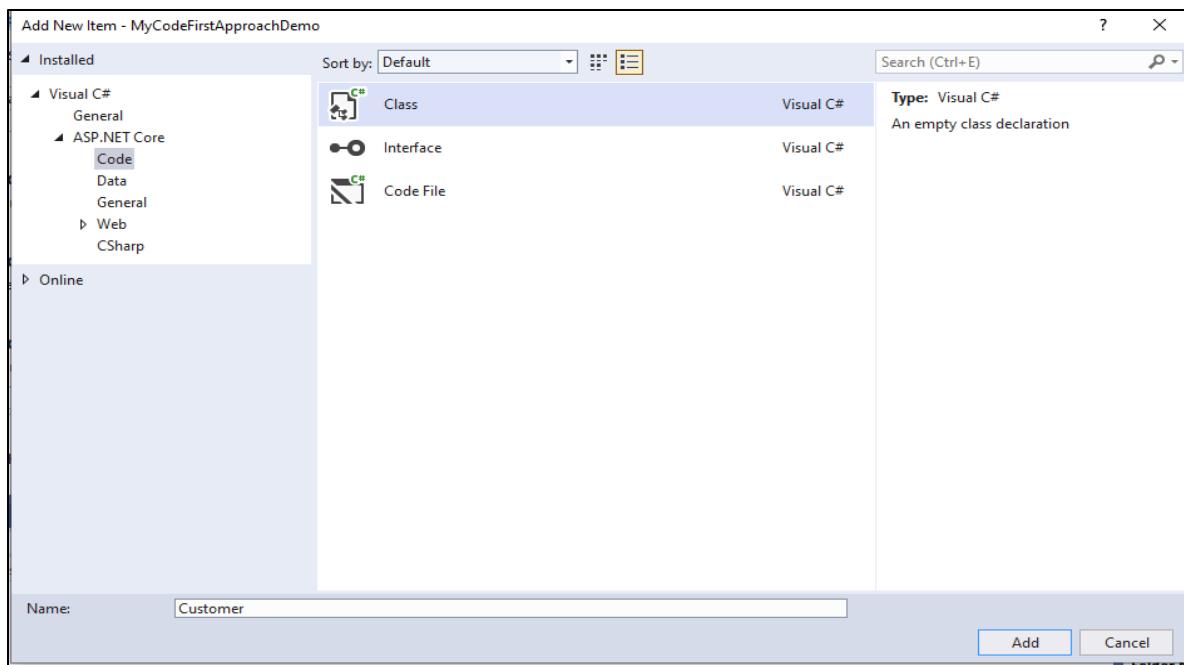
**Note:** It is important to choose the version of both these packages that are compatible with your .NET Core version.

- To add a class, from Solution Explorer, select **Models** folder, right-click, and from the shortcut menu, select **Add → Class** as shown in Figure 3.11.



**Figure 3.11: Add a Class**

12. To create a class named Customer, change the default class name to **Customer** and then, click **Add** as shown in Figure 3.12.



**Figure 3.12: Change Default Name of Class**

Default auto-generated template for the class is shown in Figure 3.13.

```
Customer.cs NuGet: MyCodeFirstApproachDemo
MyCodeFirstApproachDemo MyCodeFirstApproachDemo
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5
6  namespace MyCodeFirstApproachDemo.Models
7  {
8      public class Customer
9      {
10      }
11  }
12
```

The screenshot shows the 'Customer.cs' file in the Visual Studio code editor. The title bar says 'Customer.cs' and 'NuGet: MyCodeFirstApproachDemo'. The code editor displays the following C# code:

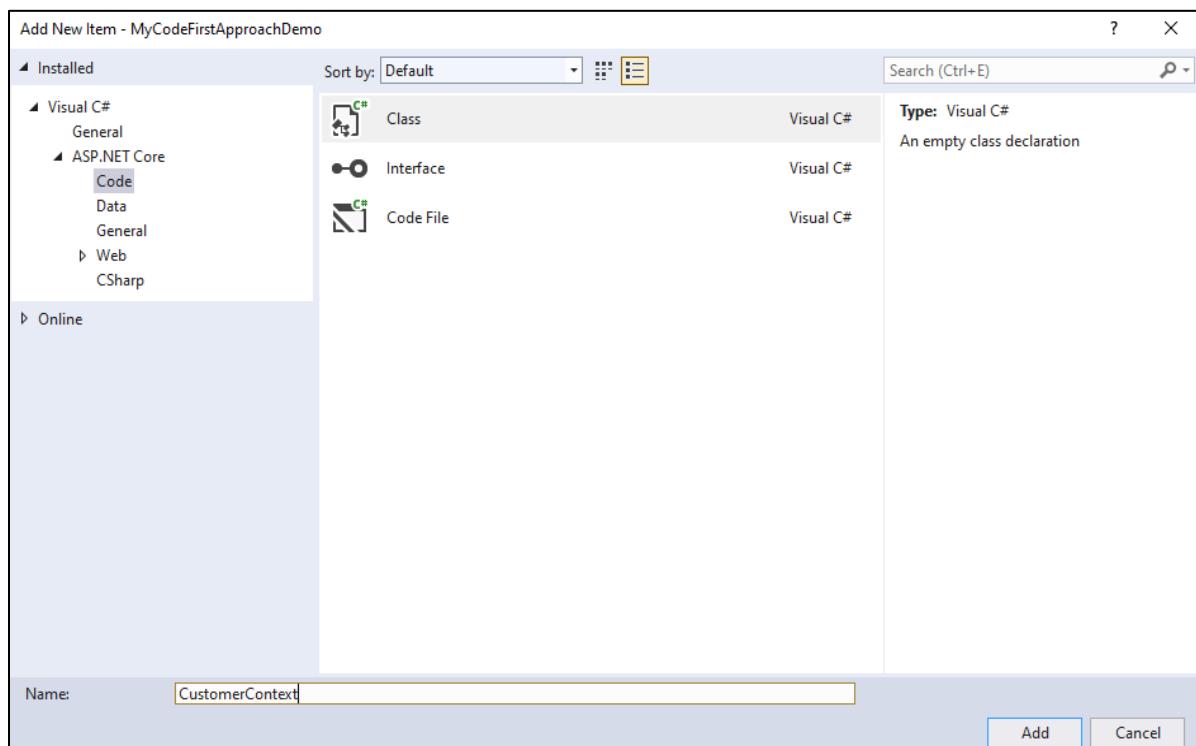
**Figure 3.13: Default Auto-generated Class Template**

13. Next, add code as per Code Snippet 1 to define properties.

### Code Snippet 1

```
public class Customer {  
    public int Id{ get; set; }  
    public string CustName { get; set; }  
    public bool IsMarried { get; set; }  
}
```

14. Next step is to add database context. Select the **Models** folder from Solution Explorer and select **Add → Class**. Enter a name for the context. In this case, the name **CustomerContext** is added as shown in Figure 3.14.



**Figure 3.14: Adding Database Context**

15. Modify the auto-generated class and add code as per Code Snippet 2.

### Code Snippet 2

```
using Microsoft.EntityFrameworkCore;  
...  
public class CustomerContext:DbContext {  
    public CustomerContext(DbContextOptions<CustomerContext>  
        options) : base(options) {  
    }  
    public DbSet<Customer> Customer { get; set; }  
}
```

While modifying the default class template for context, note following points:

- Derive a class from DbContext, which is available under using Microsoft.EntityFrameworkCore.
- Provide one constructor that is derived from the base.
- One class-based function that returns the DbSet class object.

16. Next, the connection string for the database is to be provided. Configure the **appsettings.json** configuration file to provide the connection string as shown in Figure 3.15.



```
1  {
2   "ConnectionStrings": {
3     "DefaultDatabase": "server=DESKTOP-GMKPTTF\\SQLEXPRESS;Database=SampleCore;Trusted_Connection=true;MultipleActiveResultSets=true"
4   },
5   "Logging": {
6     "LogLevel": {
7       "Default": "Information",
8       "Microsoft": "Warning",
9       "Microsoft.Hosting.Lifetime": "Information"
10    }
11  },
12  "AllowedHosts": "*"
13}
14
```

**Figure 3.15: Configuration File**

Code Snippet 3 displays the code for configuration. Note that the name of the server is (**DESKTOP-GMKPTTF\\SQLEXPRESS**). One can change the name as per local configuration. The name to be specified is the same that is given during SQL Server 2022 installation.

### Code Snippet 3

```
{
  "ConnectionStrings": {
    "DefaultDatabase": "server=DESKTOP-
      GMKPTTF\\SQLEXPRESS;Database=SampleCore;
      Trusted_Connection=true;
      MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

17. Services such as DB context should be registered with the dependency injection container later in the application. Therefore, in Solution Explorer, edit the **Program.cs** file and add the code marked in bold in Code Snippet 4.

#### Code Snippet 4

```
using MyCodeFirstApproachDemo.Models;
using Microsoft.EntityFrameworkCore;

...
public static void Main(string[] args) {
    var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<EmployeeContext>(options =>
options.UseSqlServer(
builder.Configuration.GetConnectionString("DefaultDatabase")
));
...
...
```

From .NET 6 onwards, Microsoft unified `Startup.cs` and `Program.cs` into one `Program.cs` to promote the use of minimal APIs. Two new types `WebApplication` and `WebApplicationBuilder` were introduced to make it easier to create Web applications.

The `CreateBuilder(String[])` method of `WebApplication` initializes a new instance of the `WebApplicationBuilder` class.

`WebApplicationBuilder` is responsible for following tasks:

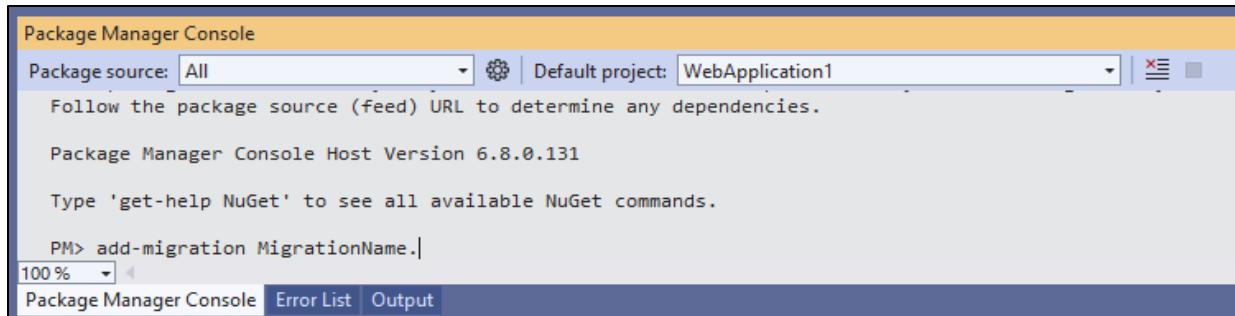
- Adding Configuration using `builder.Configuration`.
- Adding Services using `builder.Services`
- Configure Logging using `builder.Logging`
- General `IHostBuilder` and `IWebHostBuilder` configuration

It represents a builder for Web applications and services.

Hence, the code to register the `DbContext` is now added into `Program.cs`.

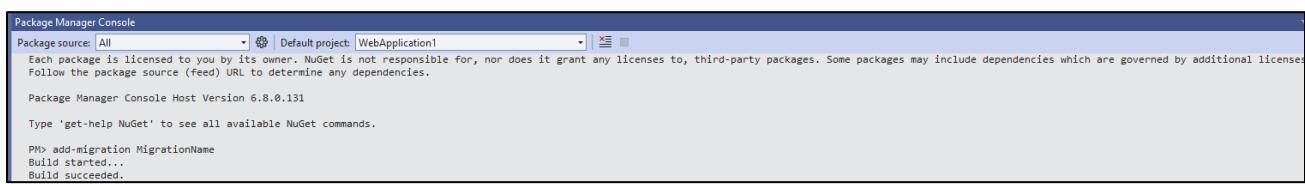
The next step is migration, which here means getting the data from SQL Server. To do the same, the Package Manager console is used.

18. Open **Package Manager Console** through **Tools → NuGet Package Manager** option in Visual Studio and type the command `add-migration MigrationName`. Press the **Enter** key as shown in Figure 3.16. In practical scenarios, `MigrationName` will be replaced by a more meaningful name.



**Figure 3.16: Migration**

Figure 3.17 displays that the command is successful and the build has succeeded.



**Figure 3.17: Build Message**

Automatically within Visual Studio, a file named **MigrationName.cs** is created as shown in Figure 3.18. It contains the definition for a class, **MigrationName**.

The migration script helps to create databases and tables in SQL Server 2022.

```

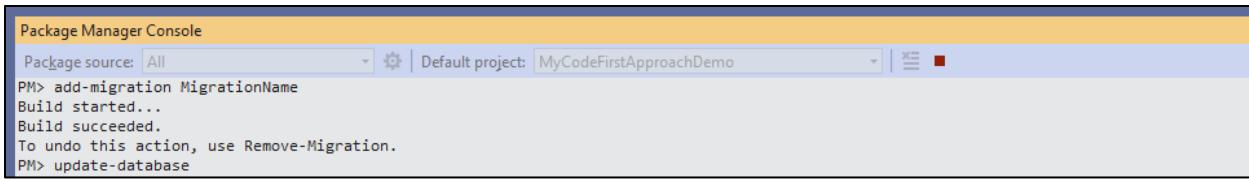
20220322040119_MigrationName.cs  X  Startup.cs  appsettings.json  CustomerContext.cs  Customer.cs  NuGet: MyCodeFirst
MyCodeFirstApproachDemo  MyCodeFirstApproachDemo.Migrations.MigrationName  Up(MigrationBuilder migrationBuilder)
1  using Microsoft.EntityFrameworkCore.Migrations;
2
3  namespace MyCodeFirstApproachDemo.Migrations
4  {
5      public partial class MigrationName : Migration
6      {
7          protected override void Up(MigrationBuilder migrationBuilder)
8          {
9              migrationBuilder.CreateTable(
10                  name: "Customer",
11                  columns: table => new
12                  {
13                      Id = table.Column<int>(nullable: false)
14                          .Annotation("SqlServer:Identity", "1, 1"),
15                      CustName = table.Column<string>(nullable: true),
16                      IsMarried = table.Column<bool>(nullable: false)
17                  },
18                  constraints: table =>
19                  {
20                      table.PrimaryKey("PK_Customer", x => x.Id);
21                  });
22          }
23      }
24  }

```

The screenshot shows the 'MyCodeFirstApproachDemo' solution in Visual Studio. The 'MigrationName.cs' file is open in the code editor. The code defines a partial class 'MigrationName' that inherits from 'Migration'. It contains a single method 'Up' which uses 'MigrationBuilder' to create a table named 'Customer' with an identity column 'Id' and a nullable string column 'CustName'. A primary key constraint 'PK\_Customer' is also defined on the 'Id' column.

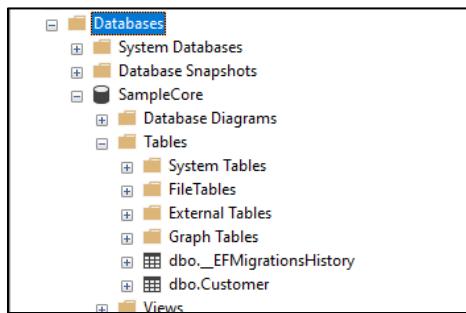
**Figure 3.18: MigrationName Class**

19. Finally, in the Package Manager Console, type the update-database command and press **Enter** as shown in Figure 3.19.



**Figure 3.19: update-database Command**

In MS SQL Server 2022, there will be a **SampleCore** database generated, and under the Tables node, **Customer** table will be created as shown in Figure 3.20.



**Figure 3.20: Database and Table Created in SQL Server 2022**

Right-click **Controllers** folder in Solution Explorer and select **Add → Controller**. Name it as CustomerController. Add code shown in Code Snippet 5 to CustomerController.cs. This will ensure the respective actions on the Web page are handled appropriately.

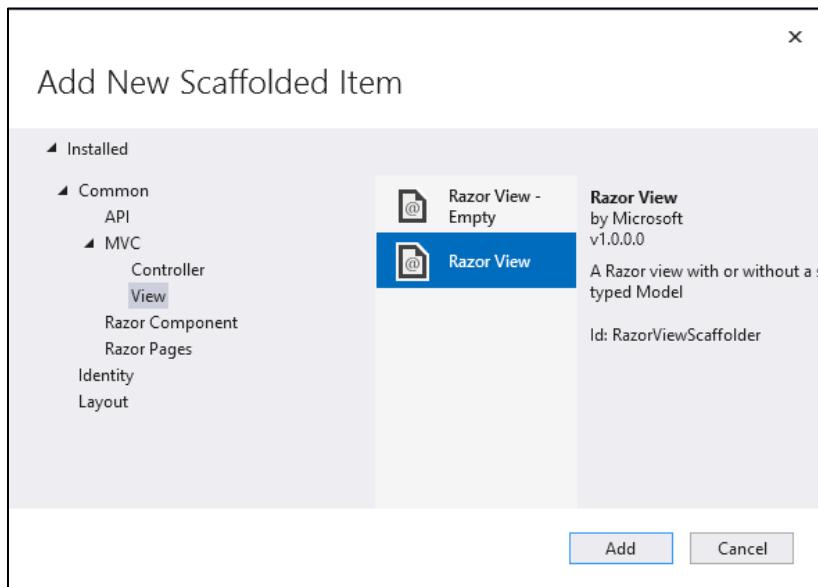
#### Code Snippet 5

```
public class CustomerController : Controller {
    CustomerContext _context;

    public CustomerController(CustomerContext context)
    {
        _context = context;
    }

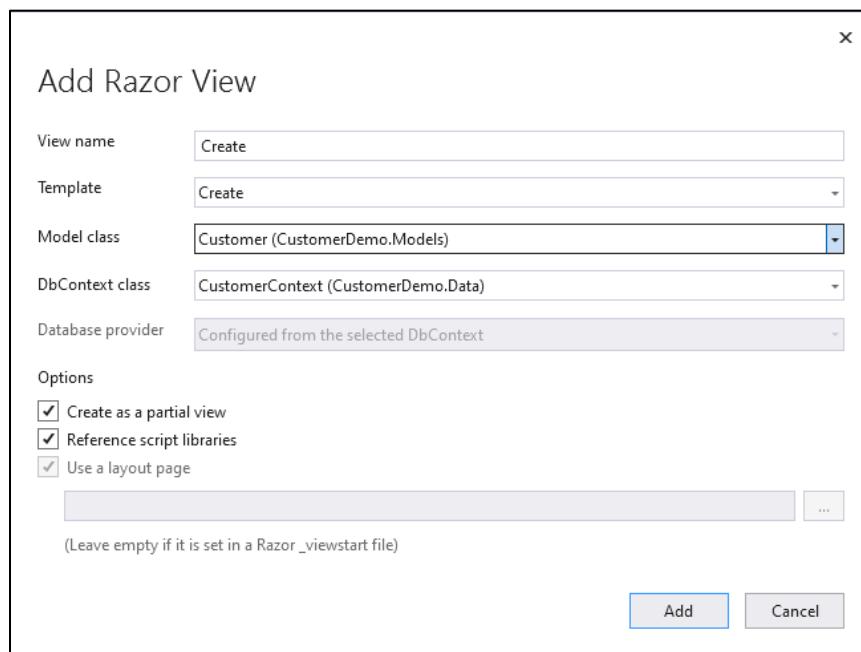
    public IActionResult Create() {
        return View();
    }
    [HttpPost]
    public ActionResult Create(Customer objCustomer)
    {
        _context.customer.Add(objCustomer);
        _context.SaveChanges();
        return View();
    }
}
```

Right-click the second Create () method and select **Add View** from the context menu. In the **Add New Scaffolded Item** dialog box, select Razor View. Refer to Figure 3.21.



**Figure 3.21: Adding a View**

Specify options from the drop-downs as shown in Figure 3.22.

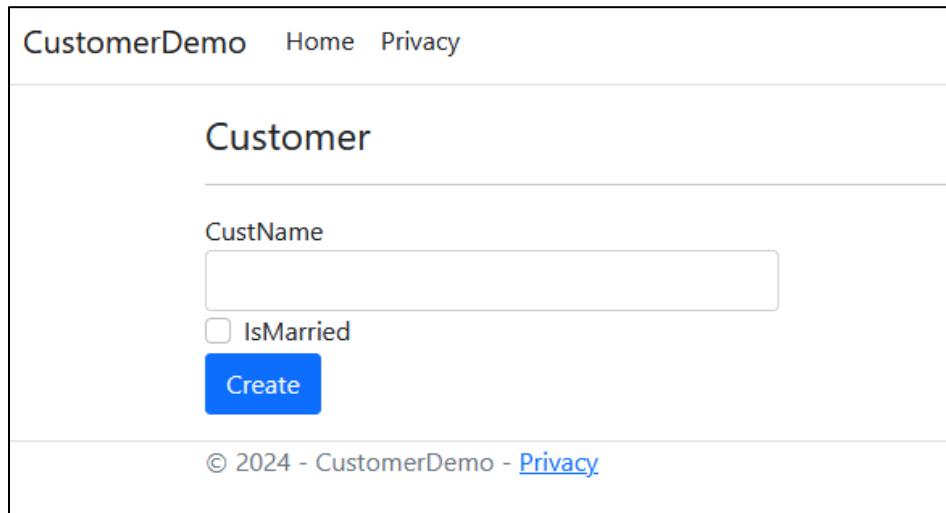


**Figure 3.22: Adding Razor View for Create**

Click **Add**. This generates a View file `Create.cshtml` under Customer subfolder of Views.

Build and run the application. Navigate to the URL:  
<https://localhost:7128/Customer/Create>

The page displays a form to enter Customer details. Refer to Figure 3.23.



The screenshot shows a web page titled "CustomerDemo" with a navigation bar for "Home" and "Privacy". The main content area is titled "Customer". It contains a form with a text input field labeled "CustName", a checkbox labeled "IsMarried", and a blue "Create" button. At the bottom, there is a copyright notice: "© 2024 - CustomerDemo - [Privacy](#)".

**Figure 3.23: Output of the MVC Application**

Upon clicking **Create**, the record is added to the SQL Server table `Customer`. Similar to Create, you can also generate Views for Edit, Update, and Delete operations.

#### Troubleshooting Tips:

1. If an error occurs saying `add-migration` command not recognized, install following package using NuGet Package Manager:  
`Microsoft.EntityFrameworkCore.Tools`
2. If you get an `InvalidOperationException`: The view <ViewName> was not found error, navigate to your project solution in Solution Explorer, right-click and add following nuget package named  
`Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`. Then, add following bolded statement in `Program.cs`:  

```
builder.Services.AddControllersWithViews().AddRazorRuntimeCompilation();
```
3. If you get an error saying "The certificate chain was issued by an authority that is not trusted" when connecting to the database, make sure you have `TrustServerCertificate=True`; although this is not recommended in a practical scenario.

# Summary

- ✓ ADO.NET is a core component of the .NET framework and is used for establishing a connection between an application and data sources.
- ✓ Data layer is the storage space where data is stored.
- ✓ RDO is an object-oriented Data Access interface that combines the simple capabilities of DAO with the low power and flexibility of ODBC.
- ✓ ADO is a layer that allows application code to access any data that is stored in a generic manner without the requirement for database implementation.
- ✓ The Entity System is an Object-Relational Mapping (ORM)-based database management framework.
- ✓ EF is an improved version of ADO.NET that provides programmers with a completely automated database interface.
- ✓ EF uses LINQ queries instead of SQL queries.
- ✓ In the code-first approach, domain classes are created first.
- ✓ EF wizard generates the database tables based on the code.

# Test Your Knowledge



1. Which of the following statements are true about ADO.NET?

<b>A</b>	It is the successor of DAO
<b>B</b>	Allows access to database without the necessity to implement database
<b>C</b>	It is based on XML
<b>D</b>	ADO.NET is a set of classes

2. EF uses \_\_\_\_\_ that are then translated to database specific queries.

<b>A</b>	LINQ queries
<b>B</b>	SQL queries
<b>C</b>	JSON data
<b>D</b>	XML data

3. Which of the following objects is better to work with for manipulating data?

<b>A</b>	SqlConnection object
<b>B</b>	SqlCommand object
<b>C</b>	SqlDataReader object
<b>D</b>	DataSet object

4. Using Entity Framework (EF), the model or entity is designed first in which of these methods?

<b>A</b>	Code-first
<b>B</b>	Model-first
<b>C</b>	Database-first
<b>D</b>	None of these

5. Which of the following components act as bridge between DataSet object and data source?

<b>A</b>	DataAdapter
<b>B</b>	Command
<b>C</b>	DataReader
<b>D</b>	Connection

## **Answers**

1	B, C, D
2	A
3	D
4	B
5	A

## **Try It Yourself**

1. Create an ASP.NET Web application in C# to interact with a database using ADO.NET.

### **Steps:**

- i. Design a simple data model (such as Person with Id, Name, and Age).
  - ii. Establish a connection to a database (you can use a local database such as SQL Server Express).
  - iii. Implement methods to:
    - o Insert a new person record into the Person table in the database.
    - o Retrieve and display all persons from the table in the database.
    - o Update the age of a person.
    - o Delete a person from the database.
2. Create an ASP.NET Core MVC application using Entity Framework for data access.

### **Steps:**

- i. Design a data model (such as Book with Id, Title, Author, and PublicationYear).
- ii. Set up Entity Framework in your ASP.NET Core MVC application.
- iii. Create a controller with actions to:
  - a. Display a list of all books.
  - b. Add a new book to the database.
  - c. Edit the details of an existing book.
  - d. Delete a book from the database.

# Session 4: Client-side Development Using ASP.NET Core MVC

## Session Overview

This session explains implementing styles, using data annotations, and routing. It also explains how to implement client-side development using ASP.NET Core MVC. It describes server-side validation.

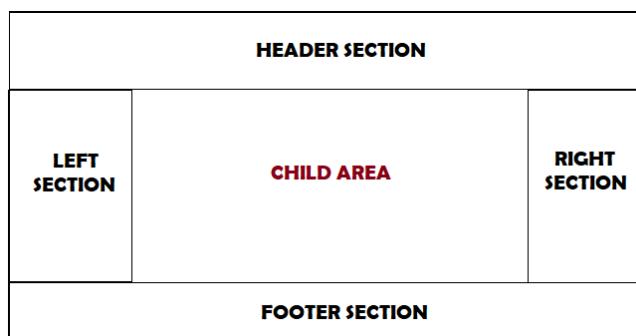
## Objectives

In this session, students will learn to:

- ✓ Describe layouts in ASP.NET Core MVC
- ✓ Explain implementing styles in ASP.NET Core MVC applications
- ✓ Explain Data Annotations
- ✓ Describe routing
- ✓ Explain dependency injection
- ✓ Identify the process to create a Single Page Application
- ✓ Describe client side validation and server side validation

## 4.1 Layout View in ASP.NET Core MVC

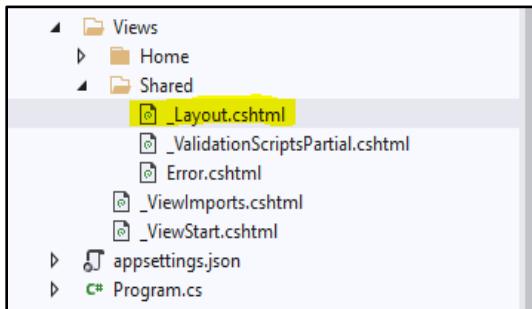
In Web applications, a specific part of the user interface is the same across all pages. These specific sections are the header section, footer section, and left and right navigation as shown in Figure 4.1.



**Figure 4.1: UI Sections**

In ASP.NET MVC, the User Interface (UI) placement mechanism called the Layout view contains some common sections. This is to ensure that the coding for these sections is not done on each page. The layout view is similar to the master page in conventional ASP.NET Web Form applications.

Layout views are shared with multiple users; therefore, it is stored in the Shared folder. For example, in the project `MyCodeFirstApproachDemo`, the Layout view file called `_Layout.cshtml` is created as shown in Figure 4.2.



**Figure 4.2: \_Layout.cshtml File**

The code of `_Layout.cshtml` is shown in Code Snippet 1.

### Code Snippet 1

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
        scale=1.0" />
    <title>@ ViewData["Title"] - MyCodeFirstApproachDemo</title>
    <link rel="stylesheet"
        href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm
            navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index"asp-area="" asp-
                                controller="Home" asp-action="Index"
```

```

        </li>
    <li class="nav-item">

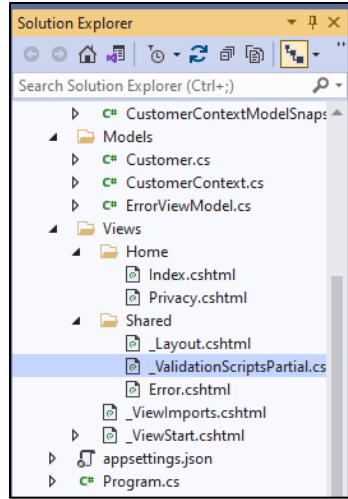
        <a class="nav-link text-dark" asp-area=""
           asp-controller="Home"
           asp-action="Privacy"asp-area=""
           asp-controller="Home" asp-action="Privacy"src="~/js/site.js" asp-append-version
        ="true"></script>
        @RenderSection("Scripts", required: false)
</body>
</html>

```

The code has collapsed section for the Header and collapsed section for the Footer and the container area. RenderBody() is called to display the child views.

## 4.2 Using Layout View

The default \_ViewStart.cshtml is included in the Views folder as shown in Figure 4.3.



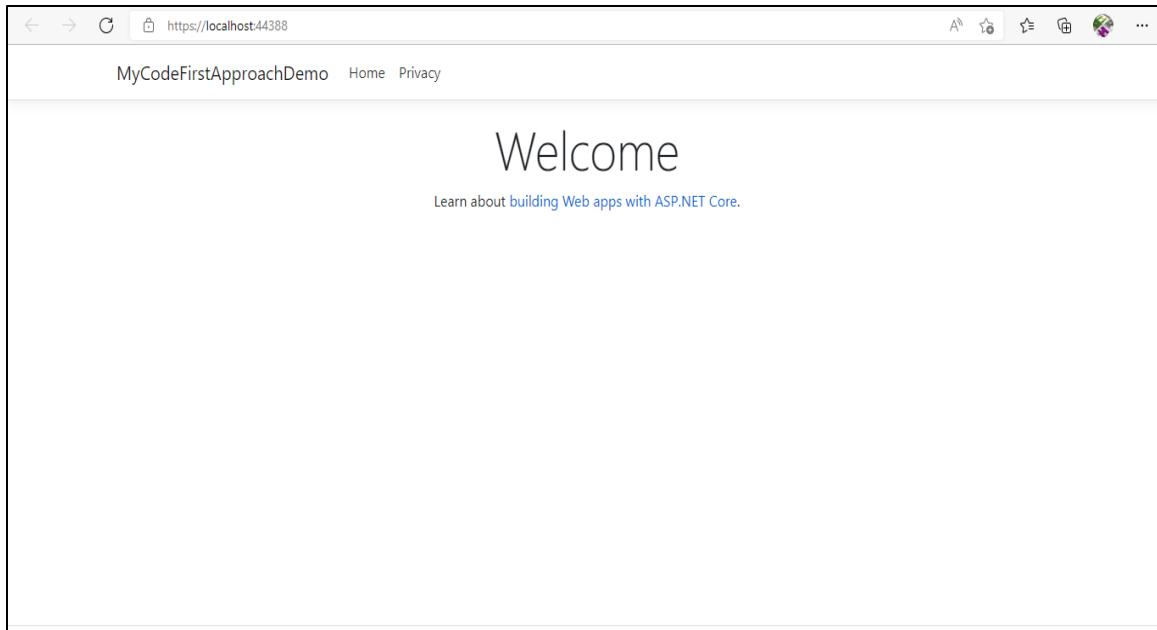
**Figure 4.3: Views Folder**

The idea behind this is to provide functionality as per views. For example, Customer related views can have different `_ViewStart.cshtml` and Stock related views can have different `_ViewStart.cshtml`. Figure 4.4 displays the code to provide `_Layout` as the default layout page.

```
_ViewStart.cshtml
1 @{
2     Layout = "_Layout";
3 }
4
```

**Figure 4.4: Default Layout**

The output of the layout is shown in Figure 4.5.



**Figure 4.5 Layout Output**

To create another layout `_LayoutVersion2.cshtml` with a change in background color, the code is changed as shown in Code Snippet 2.

### Code Snippet 2

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
    scale=1.0" />
    <title>@ViewData["Title"] - MyCodeFirstApproachDemo</title>
    <link rel="stylesheet" href =
        "~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href("~/css/site.css" />
</head>
<body style="background-color:aqua">
    <header>
        <nav class="navbar navbar-expand-sm
        navbar-toggleable-sm
        navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
                controller="Home" asp-action=
                "Index">MyCodeFirstApproachDemo</a>
                <button class="navbar-toggler" type="button"
                data-toggle="collapse" data-target=".navbar-
                collapse"
                aria-controls="navbarSupportedContent"
```

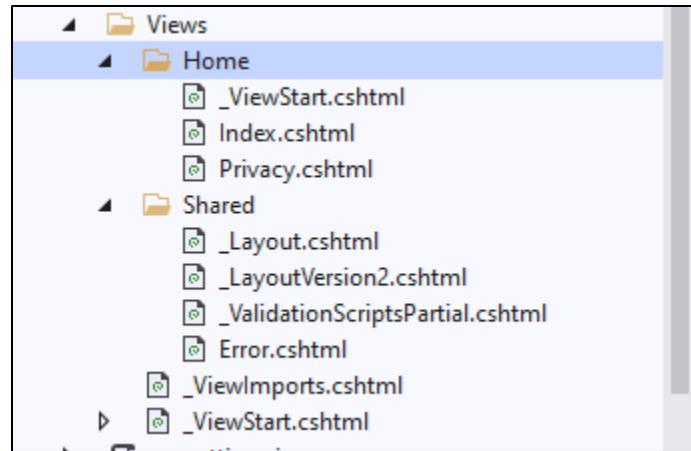
```

        aria-expanded="false" aria-label="Toggle
        navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
<div class="navbar-collapse collapse d-sm-inline-
flex flex-sm-row-reverse">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-
area="" asp-controller="Home" asp-
action="Index">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-
area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </li>
    </ul>
</div>
</div>
</nav>
</header>
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>

<footer class="border-top footer text-muted">
    <div class="container">
        © 2022 - MyCodeFirstApproachDemo -
        <a asp-area=""
            asp-controller="Home" asp-action=
            "Privacy">Privacy</a>
    </div>
</footer>
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src=
"~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-
version="true"></script>
    @RenderSection("Scripts", required: false)
</body>
</html>

```

The `_ViewStart.cshtml` can also be created in the sub-folders of the View folder to set the default layout page for all the views included in that particular subfolder as shown in Figure 4.6.

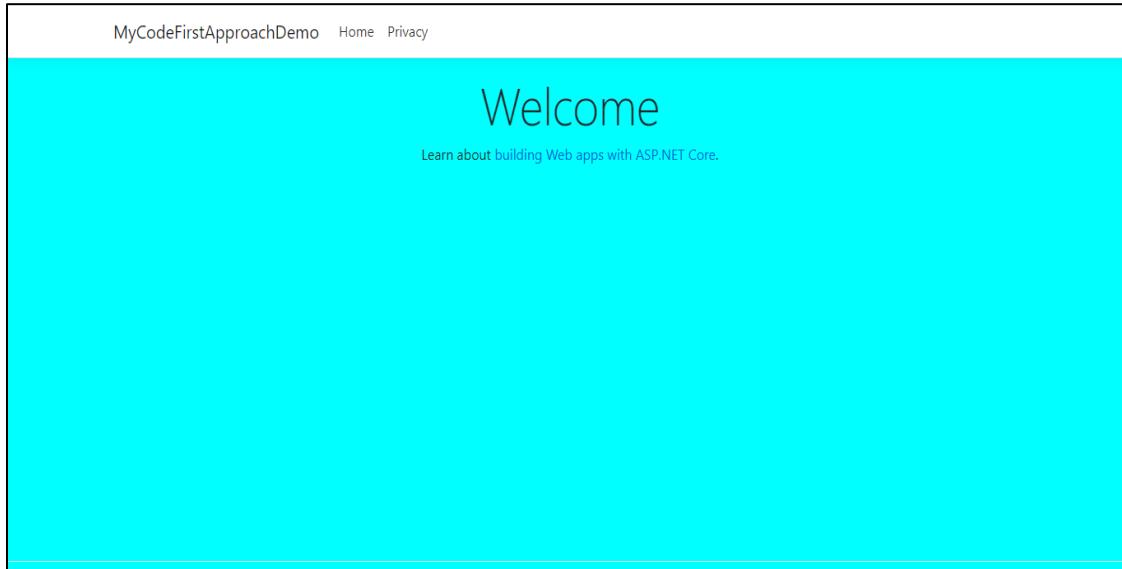


**Figure 4.6: Subfolders of Views**

In the `_ViewStart.cshtml` under the Home View folder, `_LayoutVersion2.cshtml` can be added to display the changed layout as shown in Figures 4.7 and 4.8.

```
_ViewStart.cshtml
1 @{}
2 Layout = "_LayoutVersion2";
3 }
4
```

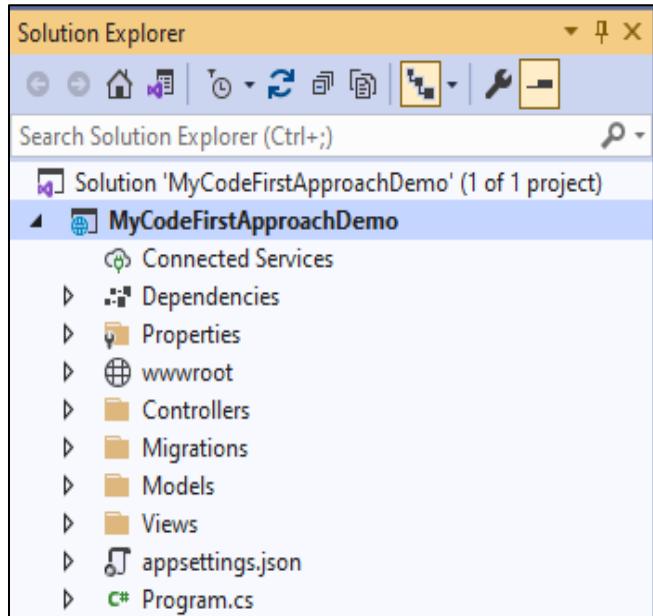
**Figure 4.7: \_ViewStart.html Layout**



**Figure 4.8: Layout with Changed Background Color**

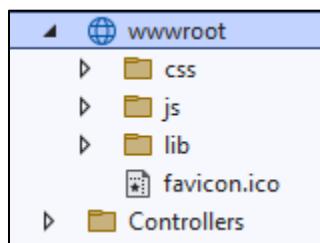
### 4.3 Implementing Styles

Consider an example to demonstrate how to implement styles in an application. In this example, the application `MyCodeFirstApproachDemo` is opened in Visual Studio 2022 as shown in Figure 4.9.



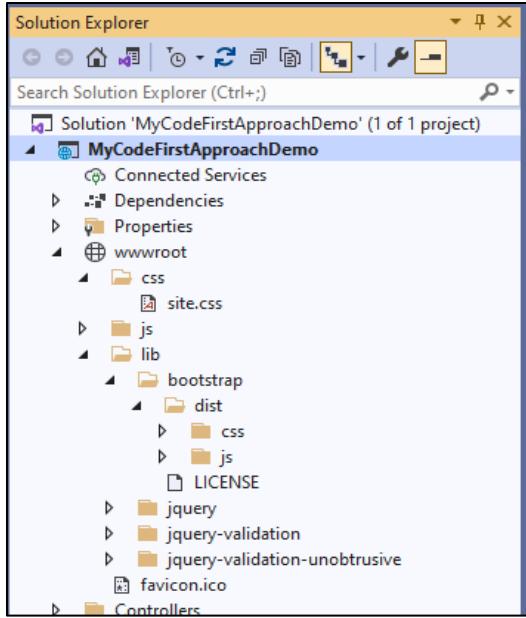
**Figure 4.9: Solution Explorer**

To open the css, js, and lib folders being used in an application, expand the **wwwroot** folder as shown in Figure 4.10.



**Figure 4.10: wwwroot Folder**

In the **lib** folder, there is a **bootstrap** folder with a **css** subfolder. Bootstrap is used to extend CSS and JS functionalities. It makes applications more responsive with a refined look and feel. In the example, the bootstrap folder also contains CSS, which is downloaded or preloaded in the application from different resources such as [getbootstrap.com](https://getbootstrap.com) or Twitter bootstrap as shown in Figure 4.11.



**Figure 4.11: Bootstrap Folder**

In this example, the css folder having site.css is used to add specifications for the application. The steps are as follows:

**Step 1:** Open the site.css file from Solution Explorer, default code in this file is as shown in Code Snippet 3.

### Code Snippet 3

```
a.navbar-brand {  
    white-space: normal;  
    text-align: center;  
    word-break: break-all;  
}  
.demo_css {  
    color: #fff;  
    background-color: #1b6ec2;  
    border-color: #1861ac;  
}  
/* Provide sufficient contrast against white background */  
a {  
    color: #0366d6;  
}  
.btn-primary {  
    color: #fff;  
    background-color: #1b6ec2;  
    border-color: #1861ac;  
}
```

**Step 2:** css class can be created to define the styles. For example, in a class demo\_css, three different styles namely, color, background-color, and border-color are specified as shown in Code Snippet 4.

#### Code Snippet 4

```
.demo_css {  
    color: #fff;  
    background-color: #1b6ec2;  
    border-color: #1861ac;  
}
```

**Step 3:** Open \_Layout.cshtml file that shows inclusion of css files as shown in Figure 4.12.

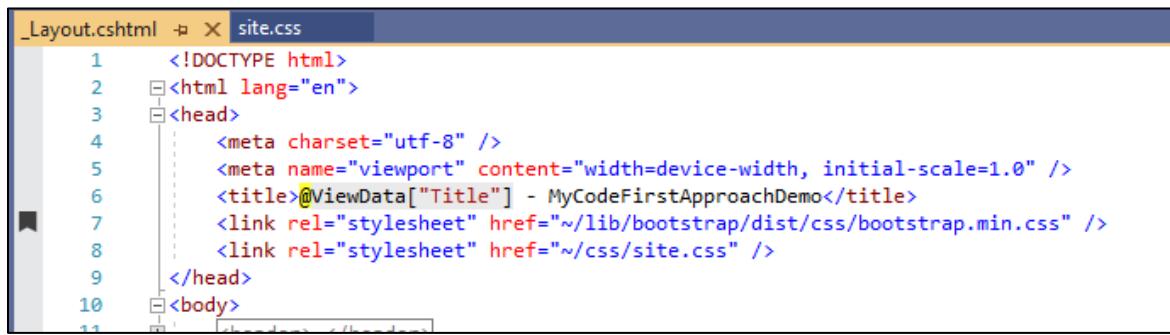


Figure 4.12: site.css

## 4.4 Data Annotations

Data validation is an important part of development, especially when dealing with big data. Data is to be analyzed using certain requirements or criteria. User also has to provide accurate data. Therefore, data validation is important.

ASP.NET MVC has a built-in namespace that has classes for data validation – System.ComponentModel.DataAnnotations.

Some of these classes are as follows:

<b>RequiredAttribute</b>	This attribute indicates that a field value is to be entered by a user.
<b>RangeAttribute</b>	This attribute indicates the numeric range in which data should fall.
<b>PhoneAttribute</b>	It indicates that the value in the field must be in a specified phone number format.
<b>UrlAttribute</b>	It allows the developer the ability to enforce specified url types. We can also make it such that users can only enter https URLs.
<b>EmailAddressAttribute</b>	This attribute is widely used to limit users to specify email addresses.

## 4.5 Routing

Routing in ASP.Net MVC is all about bridging the gap between incoming HTTP requests and forwarding them to the executable endpoints allowing the required action to be completed. Endpoints are the entry point for performing the task. The endpoint matching process can extract values from the URL of a request and offer them for processing. Routing can also produce URLs that map to the endpoints using endpoint information from the app.

Routing is a strategy that tracks requests and then, maps them to controllers and their action methods. This mapping is defined in the system and can be updated by the developer as required. Adding the Routing middleware to the request processing pipeline allows for manual mapping.

As a result, depending on the routes set in an application, the ASP.NET Core framework offers the mapping for incoming Requests or URLs to the action methods of controllers as shown in Code Snippets 5 and 6.

### Code Snippet 5

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    . . .
}
```

## Code Snippet 6

```
public IActionResult Index()
{
    return View();
}
public IActionResult Info(int id)
{
    return View();
}
```

For a single application, numerous routes can be configured. Additional information such as default settings, constraints, and message handlers can be provided on each route. A single route is created using and is named as default. A route template comparable to the default route is used by most apps with controllers and views.

Further, more routes can be added as shown in Code Snippet 7.

## Code Snippet 7

```
app.MapControllerRoute(name: "customer",
                      pattern: "customer/{*info}",
                      defaults: new { controller = "Customer", action =
"Info" } );
```

## 4.6 Dependency Injection

Dependency Injection (DI) has a very simple concept, although its implementation looks complicated. There are three terms that are related to this context. However, they are not synonyms and are as follows:

Dependency  
Injection (DI)

Inversion of  
Control (IoC)

Dependency  
Inversion Principle  
(DIP)

A design pattern where the dependencies of one object is provided by another object is DI. Dependency refers to any object or service that is used, and an injection is passing the dependency to the one who uses it (the client). A part of the client's state includes the service. The fundamental requirement of the pattern is providing the service to the client, instead of asking them to find or create a new one. One of the broader techniques of IoC is DI. DI aims to decouple objects so that even if the object that it depends on changes, the client code does not have to be changed.

The modularity of a program can be increased and made extensible using a design principle known as. IoC. Here, no new object is created by another object to complete a task. Instead, it is outsourced from an outside object. In this, the control can be inverted or reverted. The responsibility of creating objects or instances is

delegated to the code outside the class, where dependencies play a role. These objects are created by DI frameworks (also known as IoC containers).

IoC is implemented with techniques, such as factory pattern, service locator pattern, DI, and so on. In object-oriented design, decoupling software modules in a specific form is called DIP. The relationships of conventional dependency established from policy-setting modules that are at high-level, to dependency modules that are at low-levels, are reversed in this form. This renders the high-level modules independent of the low-level module implementation.

DIP states the following:

Both high-level and low-level modules should depend on abstractions.

Details should be dependent on abstractions and not vice versa. When using this principle, ensure to make use of the interfaces.

When developing ASP.NET MVC enterprise applications, developers may encounter a requirement to fuse an object into a controller so that methods of that object can be called from controller class methods. Across most cases, ASP.NET MVC constructs a controller with a parameterless function `Object () { [native code] }`, which means a class cannot be injected into it. Therefore, a Controller Dependency Injection is used as shown in Figure 4.13.

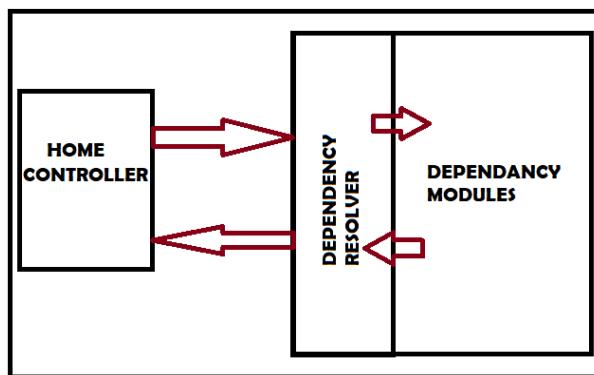


Figure 4.13: Dependency Resolver

## 4.7 Creating Single Page Applications

A Web application that fits on a single Web page is a Single Page Application (SPA). Static HTML views, Cascaded Style Sheets (CSS), and JavaScript are provided by the server in this type of application. The data is then loaded by the application through Ajax calls to the server. No postback to the server is observed while subsequent views and navigation occur. The user experiences a fluidic experience using this architecture.

As SPA loads all the data at once on one page (or on demand make various calls to receive the data in the background), the application provides no postback to the server. This makes the Web application similar to an app.

SPAs improve user experiences and are most helpful for touch screen applications. These include kiosks and touch-based point of sale systems. In these, the SPA controls 100 percent of the navigation. The only issue is that for all the initial data to load, the user has to wait for some time initially.

In traditional Web applications, the communication is initiated with the server by requesting a page. The request is then processed by the server and the HTML of the page is sent to the client. On navigating to a link or submitting a form with data, the server receives a new request and the process starts again. The request is processed by the server and a new page, in response to the new action, is sent to the browser.

After the initial request in SPA, the entire page is loaded in the browser. Ajax request takes care of the subsequent interactions. Only the portion of a page that has a change will be updated by the browser. The time taken by the application for responding to user action is reduced in SPA.

SPA involves certain challenges when compared to traditional Web applications. However, SPAs can be redesigned and rebuilt by technologies such as ASP.NET Web API, JavaScript frameworks, such as AngularJS and new styling features provided by CSS3.

Following are the advantages of SPA over multiple page applications:

Throughout the lifespan of an application, most resources (HTML, CSS, and Scripts) are loaded only once. It is only the data that is transmitted back and forth. This makes SPA faster.

It offers simplified and streamlined development. Rendering pages on the server do not require codes to be written. Development can start from a file `http://file://URI`, without using any server at all.

Monitoring network operations, investigating page elements, and data associated with it makes SPAs easy to debug with Chrome.

The same backend code can be used for Web application and native mobile application.

Local storage can be cached effectively. When an application sends one request, it is stored, and used later even in the offline mode.

Microsoft included SPA template in the first version of ASP.NET Core and since then, has maintained and updated it. It can be integrated with other popular frameworks such as Angular, React, and Vue. These SPA framework templates include command-line interface tools, build procedures, and development servers to create their own development workflow.

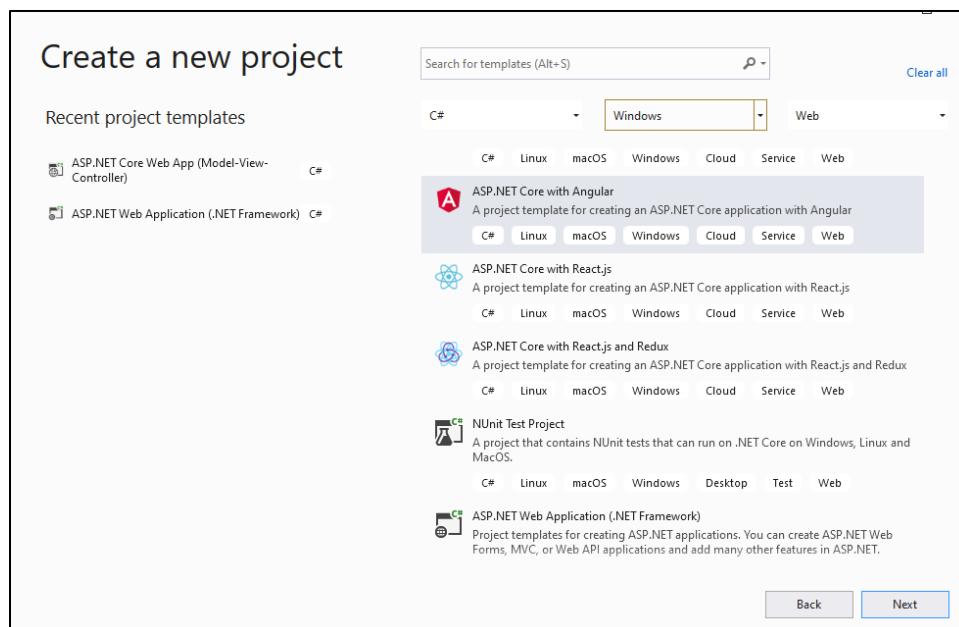
To initiate basic SPA implementation, the steps are as follows:

1. Open Visual Studio 2022 and select **Create a new project** option as shown in Figure 4.14.



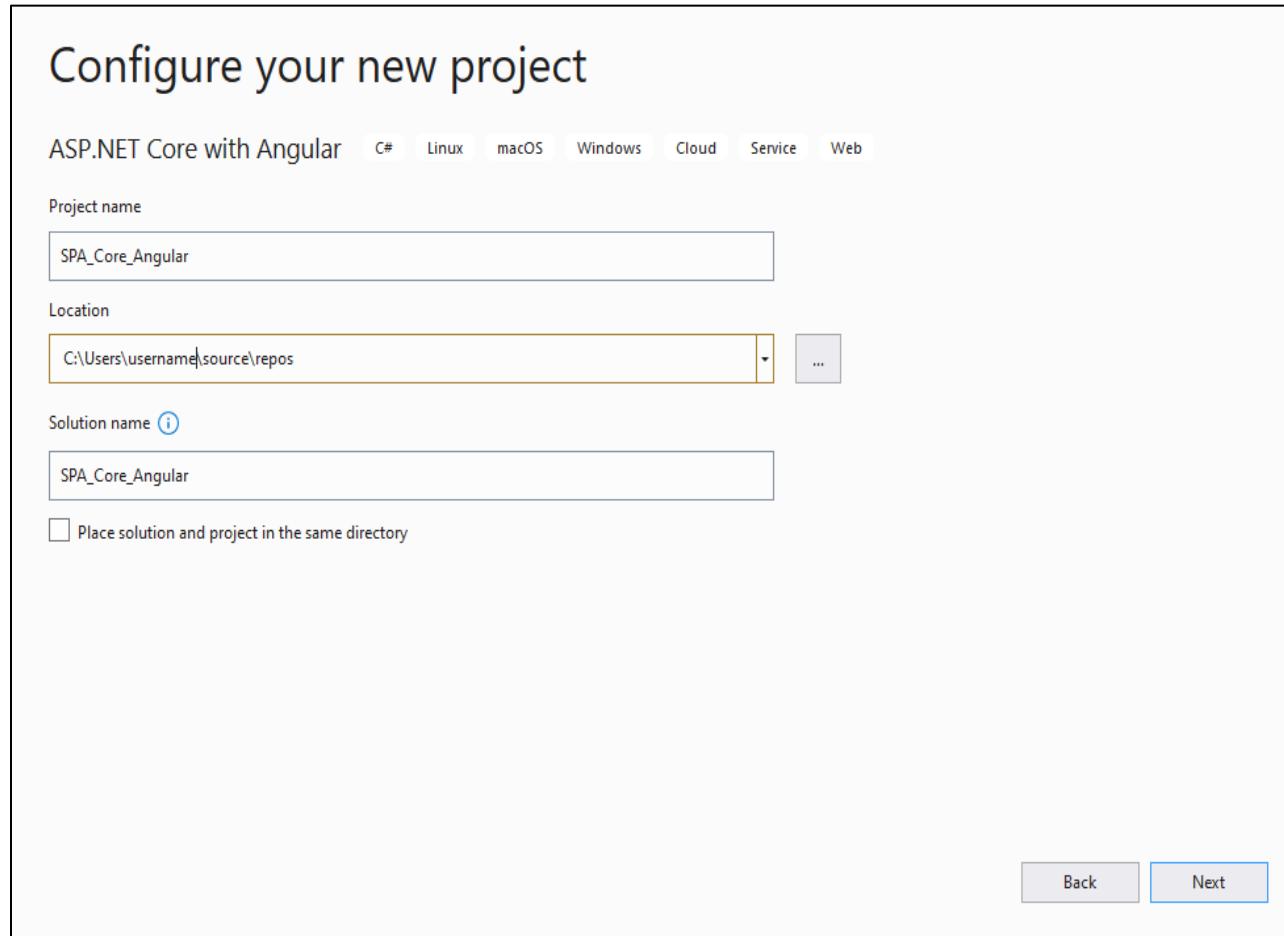
**Figure 4.14: Create a New Project**

2. Select **ASP.NET Core with Angular** option in the **Create a new project** window as shown in Figure 4.15.



**Figure 4.15: ASP.NET Core with Angular Option**

3. Next, specify the project name as **SPA\_Core\_Angular** and select the location as shown in Figure 4.16.



**Figure 4.16: Name and Location of Project**

4. Change the version of the **Target Framework** as shown in Figure 4.17.

## Additional information

ASP.NET Core with Angular C# Linux macOS Windows Cloud Service Web

Target Framework ⓘ

.NET 8.0 (Long Term Support)

Authentication Type ⓘ

None

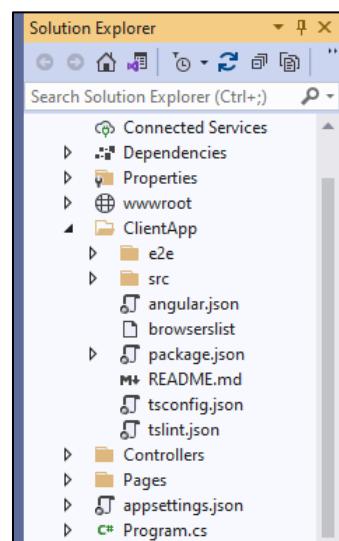
Configure for HTTPS ⓘ

Back

Create

**Figure 4.17: Additional Information**

The Solution Explorer is opened and a folder named **ClientApp** is created. This folder has its own **package.json**. Refer to Figure 4.18. Developers may now choose whether to run ASP.NET Core with SPA within Razor view or Angular in standalone.



**Figure 4.18: Solution Explorer**

5. Open **Startup.cs** from the main project to view two calls, `UseSpaStaticFiles` and `UseSpa` as shown in Figure 4.19. These functions are used to listen to requests that are triggered by the index file that Angular-CLI generates.

The screenshot shows the `Startup.cs` file in a code editor. The code is part of a class named `SPA_Core_Angular.Startup`. It contains several configuration methods for an ASP.NET Core application:

```

    49     app.UseSpaStaticFiles();
    50 }
    51
    52 app.UseRouting();
    53
    54 app.UseEndpoints(endpoints =>
    55 {
    56     endpoints.MapControllerRoute(
    57         name: "default",
    58         pattern: "{controller}/{action=Index}/{id?}");
    59 });
    60
    61 app.UseSpa(spa =>
    62 {
    63     // To learn more about options for serving an Angular SPA from ASP.NET Core,
    64     // see https://go.microsoft.com/fwlink/?linkid=864501
    65
    66     spa.Options.SourcePath = "ClientApp";
    67
    68     if (env.IsDevelopment())
    69     {
    70         spa.UseAngularCliServer(npmScript: "start");
    71     }
    72 });

```

A tooltip for the `UseAngularCliServer` method is displayed at the bottom right, showing its signature: `(extension) void Microsoft.AspNetCore.SpaServices.ISpaBuilder.UseAngularCliServer(string npmScript)`.

**Figure 4.19: Startup.cs**

Finally, programming can be done in AngularJS Web application wherein the Model, View, and Controller sections are provided in order to implement SPA.

Instead of Angular, one can also choose React or similar technologies available with ASP.NET Core to create SPAs.

## 4.8 Client Side and Server-Side Validations

Validation is an important aspect in Web applications. Validation can be performed either at the client end or at the server end.

### 4.8.1 Client-side Validation

Client-side validation provides better user experiences. All user inputs are validated in the user's browser itself in client-side validation. Script languages, such as JavaScript, VBScript, and so on perform this type of validation. For example, an error message is displayed if the user enters an invalid e-mail format. This helps users to correct each field before they submit the form.

In most cases, client-side validation is dependent on JavaScript language. Dangerous input can be bypassed and submitted to the server if the user turns OFF JavaScript. The

application cannot be protected from malicious attacks on the server resources and databases by client-side validation.

Consider a student portal Web application with student name, class, and marks. To validate the application, try to implement the client-side and server-side validations.

In server-side validation, the server does not verify the data entered by the user for the Name field. The client-side does this. This reduces the server load.

Changes in the form: To access the HTML element to display the HTML error message, add the `id` attribute to all the `span` tags. A JavaScript function is called to validate the input data when the form is submitted.

The `script` HTML element is included and a JavaScript function is created to validate the input data.

Client-side programming usually uses JavaScript, which is a high-level, interpreted language. Code Snippet 8 displays the use of JavaScript to validate the data from client-side. Assume that basic code is already created for the Student portal.

A few changes in the `ViewModel` (`Index.cshtml` file) have been made to validate the form at the client-side.

Code Snippet 8 refers to the `validateForm` JavaScript function that is called on submission of the form. The data is sent to the server if the `validateForm` function returns `true`. Else, it will not be sent. All `span` tags will have the `id` attribute added to them. This helps to identify the `span` tags and display the validation error messages. This code will be saved as `CreateStudent.cshtml`.

## Code Snippet 8

```
@page
@model StudentManagement.Pages.Student.CreateStudentModel
 @{
    ViewData["Title"] = "Create Student";
}
<h2>Create Student</h2>
<form method="post" onsubmit="return validateForm()">
    <table class="table table-bordered">
        <tr> <td>Student Name:</td> <td>
            <input asp-for="StudentName"/><br />
            <span id="validationName"
                  asp-validation-for="StudentName"
                  style="color:red"></span>
        </td></tr>
        <tr>
            <td>Student Email:</td>
            <td>
                <input asp-for="EmailId" /><br />
                <span id="validationEmail"
                      asp-validation-for="EmailId"></span>
            </td>
        </tr>
    </table>
</form>
```

```

                style="color:red"></span>
            </td></tr>
<tr>
    <td>Date of Join:</td>
    <td>
        <input asp-for="DateOfJoin" /><br />
        <span id="validationDoj"
              asp-validation-for="DateOfJoin"
              style="color:red"></span>
    </td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit" id="submitbutton"
               value="Submit" />
    </td>
</tr>
</table>
</form>

```

To validate all three fields, a JavaScript function is added. The values of all three fields are stored in separate variables. The value of each variable is verified as null or empty. The span element for the respective field is received and the text context is set with the validation error message if the value is empty. Code Snippet 9 displays these actions. Add this code in the same file.

### Code Snippet 9

```

<script type="text/javascript">
    function validateForm() {
        var isValidForm = true;
        var nameValue =
document.getElementById("StudentName").value;
        var emailValue =
document.getElementById("EmailId").value;
        var dojValue =
document.getElementById("DateOfJoin").value;

        //Validate the name field
        if (nameValue == null || nameValue == "") {
            document.getElementById("validationName")
                .textContent = "Student Name is required.";
            isValidForm = false;
        }
        //validate the class field
        if (emailValue == null || emailValue == "") {
            document.getElementById("validationEmail")
                .textContent = "Student Email is required.";
            isValidForm = false;
        }
    }
</script>

```

```

    }
    //validate the DateOfJoin field
    if (dojValue == null || dojValue == "") {
        document.getElementById("validationdoj")
            .textContent = "Date of Join is required.";
        isValidForm = false;
    }
    return isValidForm;
}
</script>

```

Client-side generates an error message when the application is run and the form is submitted without entering the data. The error message does not go to the server. The output is displayed in Figure 4.20.

**Figure 4.20: Client-side Validation Using JavaScript**

These days, in modern Web applications, developers do not manually code to perform validation at the JavaScript level. Instead, unobtrusive validation is used by most applications. In unobtrusive validation, to validate each of the fields, respective JavaScript libraries are added.

Input HTML elements have data-attributes added to them based on the data annotation attributes. This jQuery unobtrusive library receives and validates the list of fields for which data-attributes are added. A layout file (`_Layout.cshtml`) defines the layout structure of the Web application.

Unobtrusive validation is added because JavaScript libraries will be used in all the pages. JavaScript libraries are added to the layout file (`_Layout.cshtml`). This makes the JavaScript libraries available for all the View files. Code Snippet 10 displays code that is to be added before `</body>` section for unobtrusive validation.

### Code Snippet 10

```

<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.3.js">
</script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.min.js"></script>

```

```
<script src=
"https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unobtrusive.
min.js"></script>
```

Other than removing the JavaScript function, which is written for validating the fields, there is no change to the `ViewModel`. Code Snippet 11 displays the complete code for the View.

## Code Snippet 11

```
@page
@model StudentManagement.Pages.Student.CreateStudentJQModel
 @{
    ViewData["Title"] = "Create Student";
}
<h2>Create Student - Validation using JQurey</h2>
<form method="post" onsubmit="return validateForm()">
    <table class="table table-bordered">
        <tr>
            <td>Student Name:</td>
            <td>
                <input asp-for="StudentName"/><br />
                <span id="validationName"
                      asp-validation-for="StudentName"
                      style="color:red"></span>
            </td>
        </tr>
        <tr>
            <td>Student Email:</td>
            <td>
                <input asp-for="EmailId" /><br />
                <span id="validationEmail"
                      asp-validation-for="EmailId"
                      style="color:red"></span>
            </td> </tr>
        <tr>
            <td>
                Date of Join:
            </td>
            <td>
                <input asp-for="DateOfJoin" /><br />
                <span id="validationDoj"
                      asp-validation-for="DateOfJoin"
                      style="color:red"></span>
            </td>
        </tr>
        <tr>
            <td colspan="2">
```

```

        <input type="submit" id="submitbutton"
               value="Submit" />
    </td>
</tr>
</table>
</form>
```

Perform following steps for client-side validation:

1. Add the data annotations to properties in CreateStudentJQModel. Code Snippet 12 represents the validation code to be added.

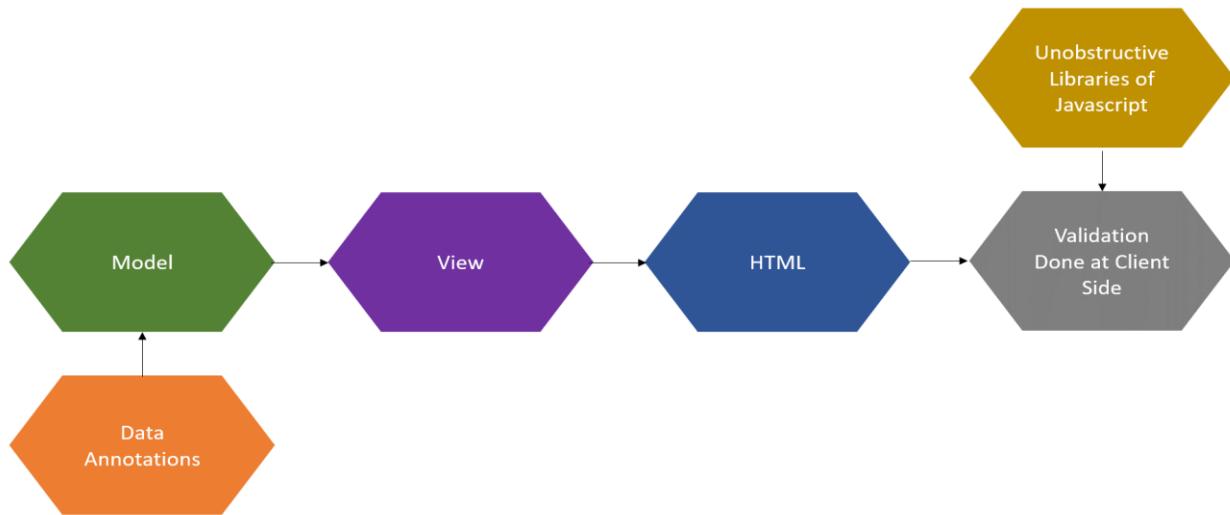
### Code Snippet 12

```

using Microsoft.AspNetCore.Mvc.RazorPages;
using System.ComponentModel.DataAnnotations;
namespace StudentManagement.Pages.Student {
    public class CreateStudentJQModel : PageModel {
        [Required]
        public string StudentName { get; set; }
        [Required]
        [DataType(DataType.EmailAddress) ]
        public string EmailId { get; set; }
        public DateTime DateOfJoin { get; set; }
        public void OnGet(){
        }
    }
}
```

2. The View model generates HTML that contains `data-*` attributes.
  - If the `Required` attribute is set for fields, an error message is created as its value, for the `data-val-required` attribute.
  - If the `MinLength` data annotation attribute is set for fields, an error message is created as its value, for the `data-val-minlength` attribute.
  - The `data-val-range` attribute is set with the error message as its value for the range data annotation. Maximum value in the range is represented by `data-val-range-max` and the minimum value in the range is represented by the `data-val-range-min` attribute.
3. Elements with `data-*` attributes are read by the jQuery unobtrusive validation library. They also perform the client-side validation. This avoids writing separate validation code using JavaScript as the configuration resolves everything.

Figure 4.21 displays this process visually.

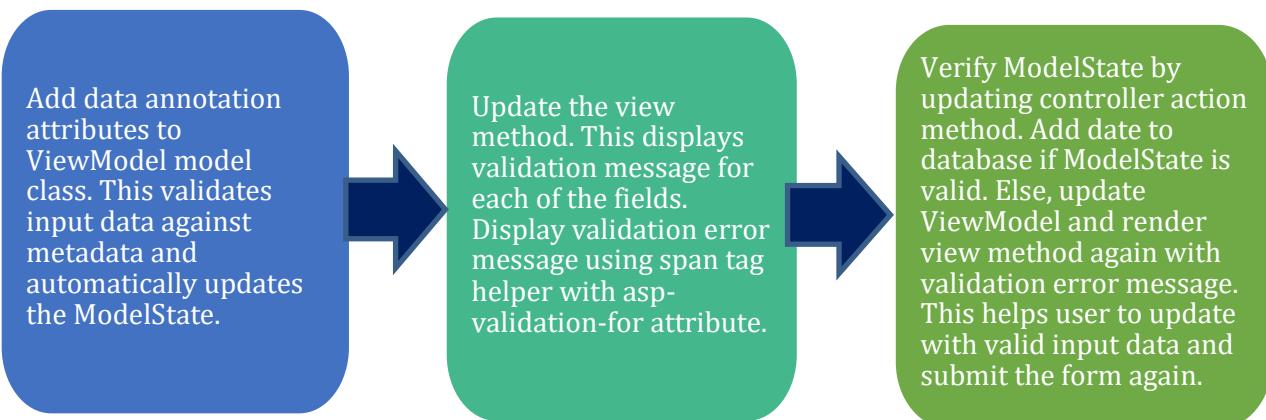


**Figure 4.21: Client-side Validation**

#### 4.8.2 Server-side Validation

Client-side validation is preferable as it is faster and is done prior to submitting the user input into the server. However, it is more vulnerable, as one can easily bypass the validation or users can submit some suspicious input that could affect the server. Thus, server-side validation is always important, safe, and secure as server validates the user input submitted by the user. In ASP.NET, this is done by server-side programming languages, such as C# or VB.NET. In case the user inputs are invalid, an error message is sent back to the Web page. It is always safe and secure to use server-side validation.

Perform following steps for server-side validation:



Data annotation attributes play an important role in case of validation rules for the properties of the model `ViewModel`. The validations fail when the input data do not match with the attribute definition in the model by using the `ModelState.IsValid` property and return as error message as `ModelState` is invalid.

Commonly used data annotation attributes are as follows:

<b>Required:</b> An indication that the property is required.	<b>Range:</b> The minimum and maximum constraints are defined.	<b>MinLength:</b> The minimum length that a property must have for the validation to succeed is defined.	<b>MaxLength:</b> The maximum length of the property is defined. The validation fails if the length of the property value exceeds.	<b>RegularExpression:</b> A regular expression for data validation is used for this attribute.
--	---	---	---	---

Include the namespace of the data annotation attribute. The data annotation attribute is available in the System.ComponentModel.DataAnnotations namespace.

Code Snippet 13 displays the updated ViewModel code.

### Code Snippet 13

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Validation.Models;
namespace Validation.ViewModels {
    public class StudentAddViewModel {
        [Required(ErrorMessage ="Student Name is required")]
        public string Name { get; set; }
        [Required(ErrorMessage ="Student Class is required")]
        [MinLength(5, ErrorMessage = "Minimum length of Class should be 5 characters")]
        public string Class { get; set; }
        [Required]
        [Range(0,100)]
        public decimal Marks{ get; set; }
    }
}
```

All three properties — Name, Class, and Marks have data annotation attributes added to them. When a validation fails, the ErrorMessage attribute displays a message. When a failure of validation occurs and no ErrorMessage is displayed, the default error message is displayed.

A span tag is added for each of the fields. This displays an error message in red color when the validation fails. No error message is displayed if the validation succeeds. The field name for which the validation error message has to be displayed is represented by the attribute value of asp-validation-for. For example, use the

span tag with the `asp-validation-for` attribute and the value `Name`. This allows the ASP.NET MVC to display the validation error message for the `Name` field. Code Snippet 14 displays these actions.

#### Code Snippet 14

```
<form asp-controller="Student" asp-action="Index">
    <table>
        <tr>
            <td><label asp-for="Name"></label></td>
            <td><input asp-for="Name" /></td>
            <td><span asp-validation-for="Name"
                style="color:red"></span></td>
        </tr>
        <tr>
            <td><label asp-for="Class"></label> </td>
            <td><input asp-for="Class" /></td>
            <td><span asp-validation-for="Class"
                style="color:red"></span>
                </td>
        </tr>
        <tr>
            <td><label asp-for="Marks"></label></td>
            <td><input asp-for="Marks" /></td>
            <td> <span asp-validation-for="Marks"
                style="color:red"></span> </td>
        </tr>
        <tr>
            <td colspan="2"><input type="submit" id="submitButton"
                value="Submit" /></td>
        </tr>
    </table>
</form>
```

Based on the data annotation attribute specified on the `ViewModel` and input data, the `ModelState` is automatically updated. In the `Index` method, verify whether the `ModelState` is valid. This is a `POST` action method. The entered data is saved in the database if the `ModelState` is valid (when the validation succeeds). The `ModelState` is set to invalid automatically if the validation fails. The `ViewModel` is then updated with the entered data and the `View` method is rendered again. This helps to correct the input data and re-submit the data by the user. Code Snippet 15 displays how the `ModelState` is updated based on the data annotations attribute.

#### Code Snippet 15

```
[HttpPost]
public IActionResult Index(StudentAddViewModel
studentAddViewModel) {
    if (ModelState.IsValid) {
```

```

using (var db = new StudentDbContext()) {
    Student newStudent = new Student {
        Name = studentAddViewModel.Name,
        Class = studentAddViewModel.Class,
        Marks = studentAddViewModel.Marks
    };
    db.Students.Add(newStudent);
    db.SaveChanges();
    //Redirect to get Index GET method
    return RedirectToAction("Index");
}
using (var db = new StudentDbContext())
{
    studentAddViewModel.StudentsList = db.Students.ToList();
}
return View(studentAddViewModel);
}

```

Error messages are displayed near the fields when the form is submitted without entering the values. Once the application starts running after making the required changes, the error is displayed.

Note following points in Code Snippets 13, 14, and 15 that display validation and error message:

Displays one error message at a time when there is more than one validation for a field. For example, validations for `Class` field are the `Required` and `MinLength` attributes. Required field error message is displayed when no data is entered for that particular field. The second validation error message is displayed when the required field error is resolved (by entering some characters in the field).

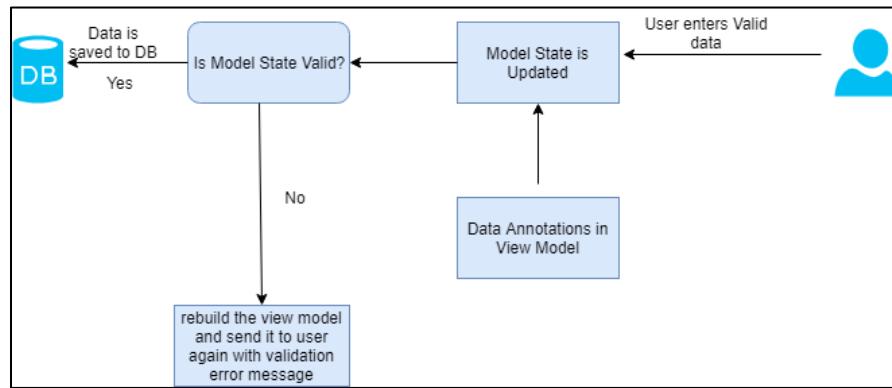
Displays the default error message when no error message is available and if the validation fails. No error message is given for the `Marks` field. Therefore, ASP.NET MVC displays the default error message when the validation fails for that field. This is based on the field name and the validation failure type.

Following steps display the high-level sequence of events in the server-side validation:



Step 2 occurs when the data in the view method is mapped to the data in the model or ViewModel during the model binding process. Rendering ViewModel helps the user to correct the input data and send the form again.

Figure 4.22 displays the server-side validation.



**Figure 4.22: Server-side Validation**

Input controls inside a form are validated both server-side and client-side (Web browser). However, client-side validation does not require a postback.

In client-side validation, the user input validation takes place on the client-side (Web browser). While in server-side validation, the user input validation takes place on the server-side during a postback session.

Client-side validation is used if no input has to be validated by the server resources for the user request. Server-side validation is used when the server resources have to validate the user input for the user request.

# Summary

- ✓ In Web applications, a specific part of the user interface is the same across all pages.
- ✓ The common sections are the header section, footer section, and left and right navigation.
- ✓ Data validation is an important part of development, especially when dealing with big data.
- ✓ Routing in ASP.Net MVC refers to bridging the gap between incoming HTTP requests and forwarding them to the app's executable endpoints.
- ✓ ASP.NET MVC has a built-in namespace `System.ComponentModel.DataAnnotations` that has classes for data validation.
- ✓ In server-side validation, the server validates the input submitted by the user. Post validation, a dynamically generated new Web page sends the feedback back to the client.
- ✓ In client-side validation, all user inputs are validated in the user's browser itself.
- ✓ A design pattern where the dependencies of one object is provided by another object is DI.
- ✓ A Web application that fits on a single Web page is a Single Page Application (SPA).

# Test Your Knowledge



1. Which of the following is a consequence of using Dependency Injection?

<b>A</b>	Loosely coupled
<b>B</b>	Modular and extensible
<b>C</b>	Complex code
<b>D</b>	Readable code
2. When is a client-side validation used?

<b>A</b>	If the input has to be validated by the browser
<b>B</b>	If the input has to be validated by the server
<b>C</b>	If no input has to be validated by the browser
<b>D</b>	If no input has to be validated by the server
3. Identify the sequence of steps occurring in the server-side validation.
  - a. Updates the ModelState.
  - b. Renders the ViewModel, if required again.
  - c. The user enters data.
  - d. Verifies the ModelState.
  - e. Saves the entered data.

<b>A</b>	c, b, a, d, e
<b>B</b>	b, d, a, c, e
<b>C</b>	c, a, d, e, b
<b>D</b>	b, a, d, c, e
4. Which of the following is an advantage of SPA over multiple page applications?

<b>A</b>	SPAs are fast as most resources are loaded only once
<b>B</b>	Development is simplified and streamlined
<b>C</b>	Easy to debug with Chrome
<b>D</b>	All of these
5. When the request matches the pattern, which of the following defines a URL pattern and a handler to use?

<b>A</b>	Annotations
<b>B</b>	Routing
<b>C</b>	Dependency Injection
<b>D</b>	None of these

## Answers

1	C
2	D
3	C
4	D
5	D

## **Try It Yourself**

1. Create a new ASP.NET Core project with an SPA template (Angular, React, or Vue). Follow the steps to run the application and explore the integration of ASP.NET Core with the chosen SPA framework.
  
2. Create a simple ASP.NET Core MVC project with a form for student registration. Add data annotation attributes to the model properties for Name, Email, and Date of Join. Implement server-side validation in the controller and display error messages if validation fails.

# *Session 5: More on ASP.NET MVC and Core MVC*

## **Session Overview**

This session describes ASP.NET Model- View-Controller (MVC) and explains its lifecycle, pattern, and advantages. It explains the three basic components of ASP.NET MVC namely, Model, View, and Controller.

## **Objectives**

In this session, students will learn to:

- ✓ Explain Role-based and View-based authorization
- ✓ Describe ASP.NET Selectors
- ✓ Explain ASP.NET Helpers
- ✓ List Action Filters
- ✓ Identify usage of Apply Action Filter and Custom Filter
- ✓ Explain ASP.NET MVC Security
- ✓ Describe Views and Partial Views in MVC

## **5.1 Role-based and View-based Authorization**

Authorization indicates the functions that can be performed by a user. It is the process of verifying whether an authenticated user has the authority or permission to access a specific resource. After creating users who can access the application, developers must specify authorization rules for the users.

For instance, a user Paul can create a document library, edit and delete documents, and add documents. On the other hand, user Tom can only read documents in a single library.

When the user accesses a resource, basic checks are done to evaluate the properties and identity for that user. Authorization in ASP.NET Core is a blend of a Policy-based model and a declarative role. Thus, authorization is indicated in prerequisites and is verified against claims submitted by the user.

ASP.NET Core authorization namespace integrates the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes that are part of authorization components.

The `AuthorizeAttribute` attribute and its different parameters control the authorization in MVC.

When the `AuthorizeAttribute` attribute is applied to an action or a controller, it restricts access to the action or the controller to any authenticated user.

Code shown in Code Snippet 1 restricts access to `LoginController` only to authenticated users.

## Code Snippet 1

```
[Authorize]
public class LoginController : Controller {
    public ActionResult Signup() {
        }
}
```

Developers can apply `Authorize` attribute to an action whenever there is requirement for authorizing an action instead of applying to an entire controller. An example of this is shown in Code Snippet 2.

## Code Snippet 2

```
public class LoginController : Controller {
    public ActionResult Signup() {
        }
    [Authorize]
    public ActionResult GetData() {
        }
}
```

`AllowAnonymousAttribute` attribute can be used to permit access by non-authenticated users to individual actions. Refer to Code Snippet 3.

## Code Snippet 3

```
[Authorize]
public class RegController : Controller {
    [AllowAnonymous]
    public ActionResult Login(){
        }
    public ActionResult Logout(){
        }
}
```

Thus, only verified users are permitted to the `RegController` excluding the `Login` action that all users can access. It is irrespective of the status such as authenticated/unauthenticated or anonymous.

### 5.1.1 Role-based Authorization

An identity of a user can belong to one or more roles. For instance, Maria could belong to the administrator and user roles whereas, Tom could belong only to the user role. It depends on the backup of the authorization process how these roles are created and managed. Developers can use `IsInRole` property on the `ClaimsPrincipal` class for the desired roles depending upon the requirement.

## Adding Role Checks

Role-based authorization checks can be inserted by developers in their code. This is done against a controller or an action in a controller. This announces roles of which the existing user must be a member in order to get access to the target resource.

The code shown in Code Snippet 4 limits access to certain actions based on specific roles.

### Code Snippet 4

```
[Authorize(Roles = "Developer")]
public class DeveloperController : Controller {
}
[Authorize(Roles = "TeamLead, Devops")]
public class Operations Controller: Controller {
}
```

Only the member users of the Devops role or the TeamLead role have access to this controller. When various attributes are applied, then a user who is utilizing the role is supposed to be a member of all the specified roles. A user has to be a member of Operator and MainUser roles, as shown in Code Snippet 5.

### Code Snippet 5

```
[Authorize(Roles = "Operator")]
[Authorize(Roles = "MainUser")]
public class SoftwareAccessController : Controller
{
}
```

Access can be further restricted by using more role authorization attributes at the action level, as shown in Code Snippet 6.

### Code Snippet 6

```
[Authorize(Roles = "Developer, Operator")]
public class SoftwareAccessController: Controller
{
    public ActionResult SetTime()
    {
    }
    [Authorize(Roles = "Developer")]
    public ActionResult ShutDown()
    {
    }
}
```

The members of the `Developer` role or the `Operator` role can access the controller and the `SetTime` action; however, only members of the `Developer` role can access the `Shutdown` action.

### Policy-based Role Checks

Policy syntax can be utilized to define role requirements. A developer executes a policy at the initial stage that is included in authorization service configuration.

This generally happens in `ConfigureServices()` in the `Program.cs` file. Refer to Code Snippet 7.

#### Code Snippet 7

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder => {
            webBuilder.ConfigureServices(services => {
                services.AddMvc();
                services.AddAuthorization(options => {
                    options.AddPolicy("RequireDeveloperRole", Policy =>
                        policy.RequireRole("Developer")));
                });
            }
        })
```

Use `Policy` property to apply policies on the `AuthorizeAttribute` attribute, as shown in Code Snippet 8.

#### Code Snippet 8

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown() {
    return View();
}
```

For a requirement of multiple allowed roles, specify them as parameters to the `RequireRole` method, as shown in Code Snippet 9.

#### Code Snippet 9

```
options.AddPolicy("ElevatedRights", Policy =>
    Policy.RequireRole("Developer", "Operator",
        "DatabaseArchitect"));
```

### 5.1.2 View-based Authorization

Typically, a developer would desire to indicate, modify, or hide a current user identity-based UI. Developers can access authorization service in MVC views using dependency injection. To inject a service into a view, developers can use @inject directive. This is like adding a property to a view and populating that property using DI. Likewise, to implement authorization service in all the views at a time, developers should add the @inject directive into the `_ViewImports.cshtml` file of the Views directory.

Once the injecting has been done, developers can use the authorization service by calling the `AuthorizeAsync` method. An example of this is shown in Code Snippet 10.

#### Code Snippet 10

```
@if (await Authorization Service.AuthorizeAsync(User,
"StandardPolicy"))
{
    <p>Successfully authorized user for standard policy.</p>
}
```

## 5.2 ASP.NET Selectors

An action selector is used to determine the action method that is invoked in response to a request. The action method to be selected for handling a request is determined by the Routing engine. Properties related to action methods are known as action selectors in ASP.NET.

Action methods play a vital role in writing an action method. In general, action selectors are utilized to name an action method. There are three types of action selector attributes:

- `ActionName`
- `NonAction`
- `ActionVerbs`

### 5.2.1 ActionName

The `ActionName` attribute can be used to define an action name other than the method name, as shown in Code Snippet 11.

#### Code Snippet 11

```
public class EmployeeController : Controller {
    public EmployeeController() {
    }
    [ActionName("Locate")]
    public ActionResult GetEmployee(in id)
    {
        // get employee from the database
    }
}
```

```

        return View();
    }
}

```

## 5.2.2 NonAction

NonAction attribute spots the public method of controller class as non-action method, as shown in Code Snippet 12.

### Code Snippet 12

```

public class EmployeeController : Controller{
public EmployeeController(){
}
[NonAction]
public Employee GetEmployee (int id){
return EmployeeList.Where(s => s.EmployeeId ==
id).FirstOrDefault();
}
}

```

## 5.2.3 ActionVerbs

When there is a requirement to regulate the selection of an action method as per the HTTP request method, the ActionVerbs selector is applied. Some ActionVerbs supported by the MVC framework are `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete`, `HttpOptions`, and `HttpPatch`. Refer to Code Snippet 13 for an example.

### Code Snippet 13

```

public ActionResult Update (int Id){
var course = courses.where(e => e.coursename ==
name ).FirstOrDefault();
return View(course);
}
[HttpPost] //HttpPost method
public ActionResult Update(Course course{
    return RedirectToAction("Home");
}

```

Table 5.1 lists the HTTP methods and their uses.

Method	Description
GET	Used for fetching data from the server. Parameters are added in the query string
POST	Used for generating a new resource
PUT	Used for appending an existing resource
HEAD	Used as GET method with the only difference being that server does not throwback message body

Method	Description
OPTIONS	Used for representing a request for data for communication options supported by browsers
DELETE	Used for deleting an existing resource
PATCH	Used for updating the resource either fully or partially

**Table 5.1: HTTP Methods**

Code Snippet 14 explains various HTTP methods.

**Code Snippet 14**

```
public class EmpController : Controller {
    public ActionResult Home() {
        return View();
    }
    [HttpPost]
    public ActionResult PostAction() {
        return View("Home");
    }
    [HttpPut]
    public ActionResult PutAction() {
        return View("Home");
    }
    [HttpDelete]
    public ActionResult DeleteAction()
    {
        return View("Home");
    }
    [HttpHead]
    public ActionResult HeadAction()
    {
        return View("Home");
    }
    [HttpOptions]
    public ActionResult OptionsAction() {
        return View("Home");
    }
    [HttpPatch]
    public ActionResult PatchAction()
    {
        return View("Home");
    }
}
```

### 5.3 ASP.NET Helpers

ASP.NET supports use of helpers which are reusable components that include code and markup to perform a monotonous or complex task. An HTML helper is a procedure that returns an HTML string. The string can result and display the desired

content. For instance, use HTML helpers to get standard HTML tags such as HTML <input>, <button>, and <img> tags. There are three types of HTML helpers.

### 5.3.1 Inline HTML Helpers

The @helper tag helps to develop inline HTML helper tags in the same view. While working on the same view, these helpers can be used again. Refer to Code Snippet 15.

#### Code Snippet 15

```
@helper ListingItems(string[] items) {  
    <ol>  
        @foreach (string i in items)  
        {  
            <li>@i</li>  
        }  
    </ol>  
}  
  
<h3>Books:</h3>  
    @ListingItems(new string[]  
    { "Novels", "Fictions", "ShortStories" })  
    <h3>List of Books:</h3>  
    @ListingItems(new string[] { "fantasy", "biography"  
    "programming" })
```

### 5.3.2 Built-In HTML Helpers

The `HtmlHelper` class extension methods are referred to as built-in HTML helpers. These can be further divided into three categories.

- **Standard HTML Helpers**

The most generic HTML elements such as HTML text boxes and check boxes are provided using standard HTML helpers.

Table 5.2 displays a list of generally used standard HTML helpers.

Element	Example
TextBox	<code>@Html.TextBox("Tb1", "val")</code> Output: <code>&lt;input id="Tb1" name="Tb1" type="text" value="name" /&gt;</code>
TextArea	<code>@Html.TextArea("Ta1", "val", 5, 20, null)</code> Output: <code>&lt;textarea cols="20" id="Ta1" name="Ta1" rows="5"&gt;val&lt;/textarea&gt;</code>
Password	<code>@Html.Password("Pwd1", "val")</code> Output: <code>&lt;input id="Pwd1" name="Pwd1" type="password" value="pwd" /&gt;</code>
Hidden Field	<code>@Html.Hidden("Hdn1", "val")</code> Output: <code>&lt;input id="Hdn1" name="Hdn1" type="hidden" value="hid" /&gt;</code>
CheckBox	<code>@Html.CheckBox("Ckb1", false)</code>

Element	Example
	Output: <input id="Ckb1" name="Ckb1" type="checkbox" value="true" /> <input name="myCkb" type="hidden" value="false" />
RadioButton	@Html.RadioButton("Rb1", "val", true) Output: <input checked="checked" id="Rb1" name="Rb1" type="radio" value="opt1" />
Drop-down list	@Html.DropDownList("Ddl1", new SelectList(new [] {"Asia", "Europe"})) Output: <select id="Ddl1" name="Ddl1"> <option>A</option> <option>E</option> </select>
Multiple-select	@Html.ListBox("Lb1", new MultiSelectList(new [] {"Tennis", "Chess"})) Output: <select id="Lb1" multiple="multiple" name="Lb1"> <option>Tennis</option> <option>Chess</option> </select>

Table 5.2: HTML Helpers

- **Strongly Typed HTML Helpers**

The most common HTML components in strongly typed view can be provided using strongly typed helpers, for example, HTML text boxes and check boxes. Table 5.3 displays a general list of strongly typed HTML helpers.

Element	Example
TextBox	@Html.TextBoxFor(m=>m.MidName) Output: <input id="MidName" name="MidName" type="text" value="MidName-val" />
TextArea	@Html.TextArea(m=>m.Add, 5, 15, new{}) Output: <textarea cols="15" id="Add" name="Add" rows="5">Addvalue</textarea>
Password	@Html.PasswordFor(m=>m.Pwd) Output: <input id="Pwd" name="Pwd" type="password"/>
Hidden Field	@Html.HiddenFor(s=>m.Sno) Output: <input type="hidden" name="Secret No" value="Sno-val" />
CheckBox	@Html.CheckBoxFor(s=>s.IsAccepted) Output: <input id="Ps1" name="Ps1" type="checkbox" value="true" /> <input name="myPs" type="hidden" value="false" />
RadioButton	@Html.RadioButtonFor(s=>s.IsApproved, "val") Output: <input checked="checked" id="Br1" name="Br1" type="radio" value="value" />
Drop-down list	@Html.DropDownListFor(m => m.Occupation, new SelectList(new [] {"Student", "Employee"})) Output: <select id="Occp" name="Occup"> <option>Student</option> <option>Employee</option> </select>

Element	Example
Multiple-select	<pre>@Html.ListBoxFor(d =&gt; d.Designation, new MultiSelectList(new [] {"Devops", "Developer"})) Output: &lt;select id="Designation" multiple="multiple" name="Designation"&gt; &lt;option&gt;Devops&lt;/option&gt; &lt;option&gt;Developer&lt;/option&gt; &lt;/select&gt;</pre>

**Table 5.3: HTML Helpers**

- **Templated HTML Helpers**

HTML elements, which are required to deliver, are identified by templated helpers as per properties of the model class. Developers should use `DataType` attribute of `DataAnnotation` class to set up proper HTML elements with templated HTML helpers. Table 5.4 displays a list of general Templated HTML helpers.

Templated Helper	Example
Display	<code>Html.Display("Name")</code>
DisplayFor	<code>Html.DisplayFor(m =&gt; m. Name)</code>
Editor	<code>Html.Editor("Name")</code>
EditorFor	<code>Html.EditorFor(m =&gt; m. Name)</code>

**Table 5.4: Templated Helpers**

### 5.3.3 Custom HTML Helpers

Code Snippet 16 depicts creating static methods in a utility class to form custom helper methods. An alternative is to develop an extension method on the `HtmlHelper` class.

#### Code Snippet 16

```
public static class CustomHelpers{
    public static MvcHtmlString SubButton(this HtmlHelper helper,
    string btnTxt) {
        string btxt = "<input type=\"submit\" value=\"" + buttonText +
    "\" />";
        return new MvcHtmlString(btxt);
    }
}
```

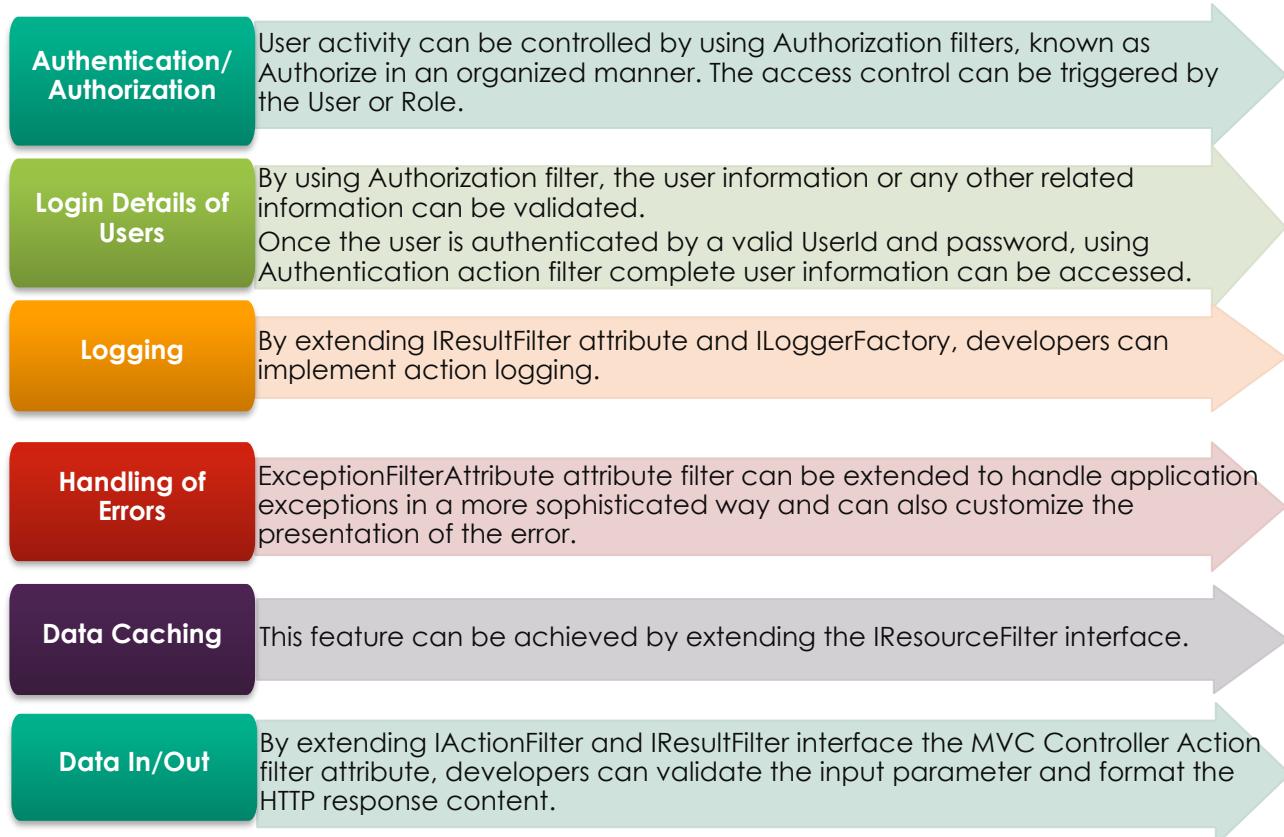
## 5.4 Introduction to Filters

ASP.NET Core filters are additional custom tasks performed prior to, post, or during execution of an action method. Filters facilitate execution of tasks that are repeated in multiple controllers or within their action methods. Filters can be implemented using ASP.NET Core MVC built-in attributes or customized attributes.

The developer can define custom attributes or filters by implementing the filter interface of ASP.NET Core MVC. Alternatively, it can be done by inheriting or overriding the methods of the available filter attribute class.

#### 5.4.1 Scenarios of When to Use Filters

Following ASP.NET MVC Core application scenarios require the use of filters:



#### 5.5 Different Types of Filters

Some of the types of filters supported by ASP.NET Core MVC are as follows:



Table 5.5 describes these filters.

Filter Type	Interfaces	Description
Authorization	IAuthorizationFilter	Controller or action limits access to the controller or action to any authenticated user
Resource	IResourceFilter	Resource filters are useful to implement caching or otherwise short-circuit the filter pipeline for performance reasons
Action	IActionFilter	Executed prior to or subsequent to intended actions

Filter Type	Interfaces	Description
Result	IResultFilter	Executed subsequent to results of actions
Exception	IExceptionFilter	Executed only if any other filter, action method, or result of actions shows an exception

**Table 5.5: Types of ASP.NET Core MVC Filters**

### 5.5.1 Authorization Filters

The authorization process determines the user's rights and all the actions that can be performed by the user. An administrative user is an example of a special type of user who can perform functions such as building document libraries and managing documents. A non-administrative user, on the other hand, has rights to only read the documents from the library.

The ASP.NET Core authorization process is a simplified procedure of determining the role for the user and is based on some strong rules set. Authorization gives rise to a set of requirements. The handlers compare these requirements against the user's formal requests. Basic checks can be made based on the rules set when the user's rights based on the identity is compared with the resource the user tries to access. Components of the Authorization filter, which includes `AuthorizeAttribute` and `AllowAnonymousAttribute`, are part of the `Microsoft.AspNetCore.Authorization` namespace.

Code Snippet 17 represents how both `Authorize` and `AllowAnonymous` attributes are used to limit access to the `StudentController` to any authenticated user.

#### Code Snippet 17

```
namespace FilterSample.Controllers{
    [Authorize]
    public class StudentController : Controller {
        [HttpGet]
        [AllowAnonymous]
        public IActionResult Index() {
            var students = new List<Student>();
            using (var dbContext = new
                StudentDbContext()) {
                students =
                dbContext.Student.ToList();
            }
            return View(students);
        }
        [HttpGet]
        Authorize(Roles = "Administrator")]
        public IActionResult Delete(int? id) {
            Student student;
            using (var dbContext = new
                StudentDbContext()) {
```

```

        student = dbContext.Student
        .FirstOrDefault(e => e.ID == id);
    }
    return View(student);
}
[Authorize(Roles = "Administrator")]
[HttpPost]
public IActionResult Delete(int id)
{
    Student student;
    using (var dbContext = new
    StudentDbContext())
    {
        student = dbContext.Student
        .FirstOrDefault(e => e.ID == id);
        dbContext.Student.Remove(student ??
        throw new InvalidOperationException());
        dbContext.SaveChanges();
    }
    return RedirectToAction("Index");
}
}
}

```

### 5.5.2 Resource Filters

Resource filters take care of the request post authorization. They can run code before and after the rest of the filter is executed, before the model binding happens. Resource filters are commonly used to implement caching.

They implement either the `IResourceFilter` or `IAsyncResourceFilter` interface. Before resource filters are run, only authorization filters can be run.

Resource filters are useful to short-circuit most of the work a request is doing. The ASP.NET Core request pipeline comprises request delegates, which call the next one, in the pipeline. When a delegate decides not to pass a request to the next delegate, the process is known as short-circuiting. For example, a caching filter can avoid the rest of the pipeline if the response is in the cache. Code Snippet 18 shows the built-in code for `IResourceFilter`.

#### Code Snippet 18

```

public interface IResourceFilter
{
    void OnResourceExecuted(ResourceExecutedContext context);
    void OnResourceExecuting(ResourceExecutingContext context);
}

```

### 5.5.3 Action Filters

Action filters are executed prior to an action run or subsequent to an action run. `IActionFilter` Interface provides for an action filter with a method called `OnActionExecuting`, which gets executed prior to an action, whereas the method `OnActionExecuted` gets executed subsequent to an action run. The built-in code can be seen in Code Snippet 19.

#### Code Snippet 19

```
public interface IActionFilter {  
    void OnActionExecuting(ActionExecutingContext filterContext);  
    void OnActionExecuted(ActionExecutedContext filterContext);  
}
```

### 5.5.4 Result Filters

Result filters are executed prior to or subsequent to results of actions.

Different types of Result filters are as follows:

- `ViewResult`
- `PartialViewResult`
- `RedirectToRouteResult`
- `RedirectResult`
- `ContentResult`
- `JsonResult`
- `FileResult`
- `EmptyResult`

These are all derived from the `ActionResult` class. After Action filters, are executed, Result filters are called. A Result filter is created using the `IResultFilter` interface and it provides two methods, namely, `OnResultExecuting()` and `OnResultExecuted()`.

`OnResultExecuting()` is executed ahead of an action result while `OnResultExecuted()` is executed subsequent to an action result. Refer to Code Snippet 20 for built-in code.

#### Code Snippet 20

```
public interface IResultFilter {  
    void OnResultExecuted(ResultExecutedContext filterContext);  
    void OnResultExecuting(ResultExecutingContext  
filterContext);  
}
```

### 5.5.5 Exception Filters

Exceptions are generated during execution of actions or filters. This results in calling of Exception filters. `IExceptionFilter` aids in creation of an Exception filter. The `Exception` filter allows for the `OnException` method. The `OnException` method comes into play under the condition of an exception. The execution of an action or filter can cause the condition of exception. Code Snippet 21 depicts the built-in code.

#### Code Snippet 21

```
public interface IExceptionFilter {  
    void OnException(ExceptionContext filterContext);  
}
```

### 5.5.6 Order of Filter Execution

There is a particular order in which all ASP.NET MVC Core filters are executed.

The appropriate sequence is as follows:

- Authorization filters
- Resource Filter
- Action filters
- Exception Filter
- Result filters

## 5.6 Configuring Filters

A custom filter can be configured into an application using three different levels as follows:

- **Global level**

A developer can restrict access for every Web API controller by adding the `AuthorizeAttribute` filter to the global filter list. In the `Program.cs` class, add this Code Snippet for set global authorization for all actions.

Code Snippet 22 depicts the implementation.

#### Code Snippet 22

```
services.AddMvc(config =>  
{  
    var policy = new AuthorizationPolicyBuilder()  
.RequireAuthenticatedUser()  
        .Build();  
    config.Filters.Add(new AuthorizeFilter(policy));  
});
```

- **Controller level**

This can be achieved by locating the filter on the top of the controller name. Code Snippet 23 depicts the implementation.

### Code Snippet 23

```
[Authorize]
public class StudentController : Controller {
    // All action methods of
    // Student controller are
    // secured.
}
```

- **Action level**

This can be achieved by locating the filter on top of the action name. Code Snippet 24 depicts the implementation.

### Code Snippet 24

```
[HttpGet]
[Authorize(Roles = "Administrator")]
public IActionResult Delete(int? id)
{
    // Delete action methods //secured for Admin only.
}
```

## 5.7 More on Action Filters

Action Filters are executed prior to controller action runs or subsequent to action runs.

A controller action or a complete controller can be associated with an attribute. It can be an Action filter that modifies the way the action is executed.

Action filters can also be customized. For example, a custom authentication system implementation can be done by using a custom action filter created by the developer. Another example could be that the developer wants to validate the model data coming into a controller action. This can be achieved by creating an action filter for the same.

### 5.7.1 Action Filter for Validation

Code Snippet 25 demonstrates example of `ModelValidationAttribute` Action Filter.

The example validates if the model is valid or not, in case of invalid model data, it will return as `BadRequestObjectResult`. By using this Action Filter, developers can skip model validation for all post Action Methods.

## Code Snippet 25

```
public class ModelValidationAttribute  
    : ActionFilterAttribute  
{  
    public override void OnActionExecuting  
        (ActionExecutingContext context)  
    {  
        if (!context.ModelState.IsValid)  
        {  
            context.Result = new BadRequestObjectResult  
                (context.ModelState);  
        }  
    }  
}
```

### 5.7.2 Action Filter for Handling Error

By overriding `OnActionExecuted` method and `ActionExecutedContext` developers can handle an exception in a managed way. When the controller generates an error, the application is forwarded to a custom message. Code Snippet 26 depicts the implementation.

## Code Snippet 26

```
public class OnExceptionActionAttribute  
    : ActionFilterAttribute {  
    public override void OnActionExecuted  
        (ActionExecutedContext context){  
        if ( context.Exception !=null )  
        {  
            context.Result = new ContentResult() {  
                Content = "Some critical Error  
                occurred! Try again later."  
            };  
            context.ExceptionHandled = true;  
        }  
    }  
}
```

### 5.7.3 Apply Action Filter

An Action filter refers to an attribute that uses the `ActionFilterAttribute` class. This class is an abstract class. Some Action filters are executed ahead of action method runs.

`ModelValidationAttribute` and `OnExceptionActionAttribute` are examples of Action filters and they implement `ActionFilterAttribute` class directly.

Some Action Filters implement the abstract `ActionFilterAttribute` and allow for execution of the Action filter prior to action execution by overriding `OnActionExecuting` method or subsequent to an action method by overriding `OnActionExecuted` method.

An Action filter attribute can be used to identify with any controller or action method. If an attribute is identified with a controller, the action filter is associated with all the action methods of that controller.

Code Snippet 27 demonstrates the implementation of the `ModelValidation` class.

### Code Snippet 27

```
[HttpPost]
[ModelValidation]
public ActionResult Create(STUDENT student) {
    using (var dbContext = new StudentDbContext())
    {
        dbContext.Add(student);
        dbContext.SaveChanges();
    }
    return View();
}
```

Code Snippet 28 shows an exception handled by using `OnExceptionAction` action filter attribute.

### Code Snippet 28

```
[OnExceptionAction]
public IActionResult Registration()
{
    throw new ApplicationException
        ("ActionResult not implemented.");
}
```

## 5.8 Resource Filters

Code Snippet 29 demonstrates overriding `OnResourceExecuting` method of `IResourceFilter` filter attribute. In case developers want to short-circuit any action method, they can implement this attribute.

### Code Snippet 29

```
public class PageNotAvailableAttribute : Attribute,
IResourceFilter {
    public void OnResourceExecuted(ResourceExecutedContext context)
    { }
    public void OnResourceExecuting(ResourceExecutingContext context) {
        context.Result = new ContentResult() {
            Content = "Page unavailable! Try later."
    }
}
```

```
};  
}  
}
```

### 5.8.1 Implementing Resource Filters

Code Snippet 30 depicts implementation of `PageNotAvailableAttribute` to short-circuit home/about URL.

#### Code Snippet 30

```
[PageNotAvailable]  
public IActionResult About() {  
    ViewData["Message"] = "Your application description page.";  
    return View();  
}
```

## 5.9 ASP.NET MVC Security

Security is a crucial aspect for any Web application, especially in the enterprise environment. ASP.NET and ASP.NET MVC support several robust security features.

User identities are managed with Cloud, SQL database, and local Windows active directory in the latest release of ASP.NET. Verifying the identity of a user is called as **authentication**. Access to applications must be given only to authenticated users due to obvious reasons. While starting a new ASP.NET application, configuring authentication services based on the application requirements is one of the major steps. Once the MVC template is selected, the **Change Authentication** button is enabled, as shown in Figure 5.1.

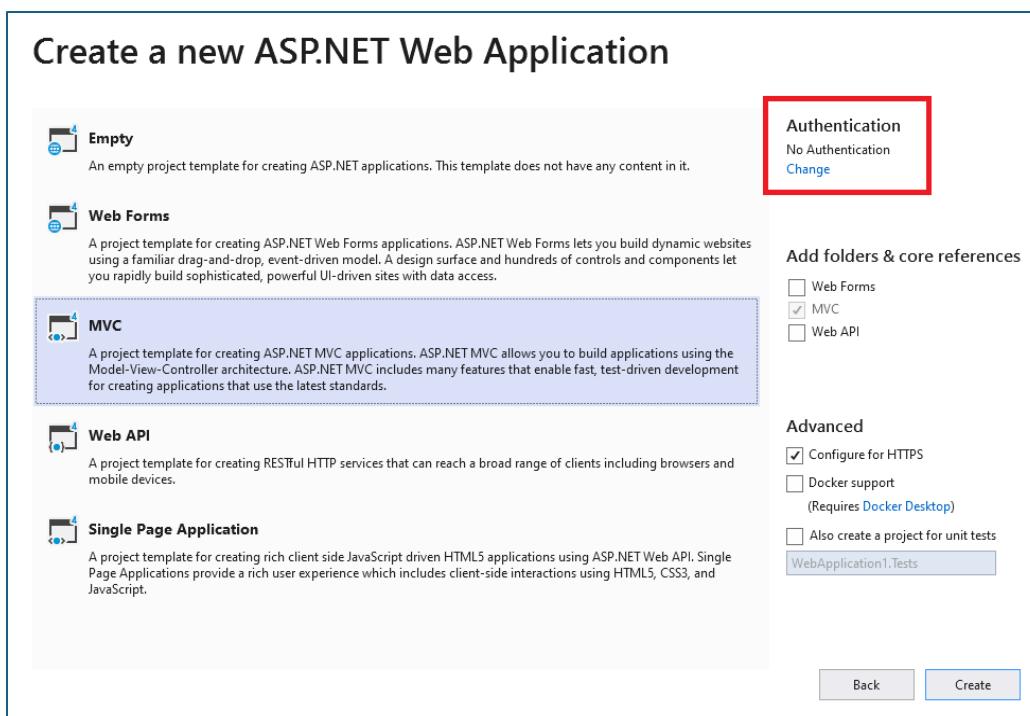
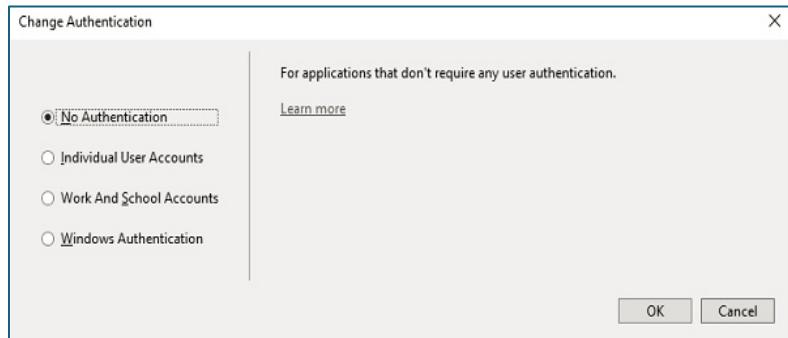


Figure 5.1: Change Authentication Button

This allows the type of authentication to be changed. **Individual User Accounts** is the default authentication. Figure 5.2 shows four options displayed when the **Change Authentication** button is clicked.



**Figure 5.2: Change Authentication Window**

Following are the characteristics of these four options:

No Authentication	Used to build a Website that does not give importance to who the visitors are and ensures that no features will be used to identify visitors to this site. However, this can also be changed later.
Individual User Accounts	Use this option for establishing the identity of the user in an Internet site. In spite of a local login and a password, third-party logins can also be enabled. These include Microsoft, Google, Facebook, and Twitter. This helps users to login using their Live account, Twitter account, or a local username. In any case, no passwords are stored.
Windows Authentication	It enables the user to launch a browser to the application within the same firewall after logging in to Windows. The user's identity is automatically picked up by ASP.NET. This option refrains anonymous access to the site. However, this configuration setting can be later changed, if required.
Work and School Accounts	Use this option for business applications, which employ active directory federation services. Office 365 is set up or Azure Active Directory Services is used. Other than these two, internal apps and Cloud apps use single sign-on.

In addition to traditional authentication methods, the latest release of ASP.NET introduces support for modern authentication protocols such as OAuth 2.0 and OpenID Connect. These protocols enable seamless integration with popular identity providers such as Microsoft Azure Active Directory, Google, Facebook, and more. By leveraging these protocols, ASP.NET MVC applications can offer Single Sign-On (SSO) capabilities, federated authentication, and secure access to external resources.

Let us now explore few other features of ASP.NET MVC.

## 5.10 Partial Views in MVC

Partial view is an exclusive view capable of rendering a portion of the view content. It is similar to a user control Web form application. Partial views can be reused in multiple views. Code duplication can be reduced using this view.

A ViewDataDictionary object, which is an own copy of the partial view, is available with the main view. The main view is also called parent view. This helps the data of the parent view to be accessed by the partial view.

The ViewDataDictionary object is used to instantiate the partial view. The data of the parent view does not get affected even if there is any change in the data (viewDataDictionary object). To render any related data in a partial view, which is part of the model, ensure to use the RenderPartial method.

Perform following steps to create a partial view:

1. Right-click **View** and select **Shared folder**.
2. Select **Add View**. Click the **Create as a partial view** check box. Name the view as TestView. Add some markup in the view.

**Note:** Creating and saving a partial view in a Shared folder is a good practice. This helps to use the partial view as a reusable component.

Add the code given in Code Snippet 31 to \_Layout.cshtml. This code uses two methods for rendering the partial view in HTML helper. These are namely, Partial() and RenderPartial().

### Code Snippet 31

```
<div>
@Html.Partial("TestView")
</div>
```

Alternatively, one can use following method to render the partial view:

```
Html.RenderPartial("TestView");
```

In the @Html.RenderPartial() method, the same `TextWriter` object that is employed by the current view is used. This means any result of this method is directly written into the HTTP response. In the @Html.Partial() method, the view is rendered as an HTML-encoded string. The result will be stored in a string variable.

The output of the `Html.RenderPartial` method is written directly to the HTTP response stream. Due to this reason, the `Html.RenderPartial()` method works slightly faster than the `Html.Partial()` method.

Code Snippet 32 displays how to return a partial view from a controller's action method.

### Code Snippet 32

```
public ActionResult PartialViewExample() {
    return NewPartialView();
}
```

Compared to a view, a partial view is more lightweight. The `RenderPartial()` method allows a regular view to be passed. A view is considered as a partial view when a layout page is not specified in the view.

However, Razor does not exhibit any difference between views and partial views as done in ASPX view engine.

Some key differences between a view and partial view are listed in Table 5.6.

<b>View</b>	<b>Partial View</b>
It includes a layout page.	It does not include a layout page.
It renders <code>viewstart</code> page before rendering any view.	It does not look for <code>viewstart.cshtml</code> . Within the <code>_viewstart.cshtml</code> page, no common code for a partial view can be placed.
It includes markup tags, such as HTML, body, head, title, meta, and so on.	It does not contain any markup as it renders within the view.

**Table 5.6: Differences Between View and Partial View**

# Summary

- ✓ MVC is a framework that helps developers to create Web applications in which sections of code are organized by the functions they perform.
- ✓ The three basic components of MVC are Model, View, and Controller.
- ✓ The two lifecycles of MVC are the application lifecycle and request lifecycle.
- ✓ The fundamental pattern component of an MVC application is the component called view. It is accountable for rendering the user interface, irrespective of it being an HTML or a UI widget on a desktop application.
- ✓ A model can be defined as a collection of classes that help developers to work with data and business logic.
- ✓ Validation forms a vital aspect in ASP.NET MVC applications. It helps developers to assess if the user input is valid.
- ✓ Controllers form the central unit of the ASP.NET MVC application. It is the first unit that any incoming HTTP Request communicates with. It also has the responsibility of deciding which model to choose.
- ✓ The MVC design pattern is definitely a better approach to create software applications. Projects created using MVC model consume less expenditure and time too.

# Test Your Knowledge



1. `RequireHttpsAttribute` is an example for which type of filter?

<b>A</b>	Result
<b>B</b>	Authentication
<b>C</b>	Action
<b>D</b>	Authorization

2. During execution of actions or filters, immediate attention is drawn by which type of filter?

<b>A</b>	Authentication
<b>B</b>	Exception
<b>C</b>	Action
<b>D</b>	Result

3. Configuring a filter at which level would allow access to all the controllers in the ASP.NET domain?

<b>A</b>	Common
<b>B</b>	Action
<b>C</b>	Controller
<b>D</b>	Global

4. What is the correct definition of Output cache?

<b>A</b>	It is a Result filter which works on the result of caching action
<b>B</b>	It is an Action filter which controls duration of caching for an output of a controller action
<b>C</b>	It is a data delivery function
<b>D</b>	It is an Exception filter

5. What is 'OnResultExecuting'?

<b>A</b>	A method type in custom action filter implementation
<b>B</b>	An attribute of Result filter
<b>C</b>	A kind of class definition in ASP.NET MVC
<b>D</b>	A process in ASP.NET framework

6. Which templated helper provides a read-only view of the defined model property?

<b>A</b>	Editor
<b>B</b>	DisplayFor
<b>C</b>	Display
<b>D</b>	EditorFor

7. Which of the following are types of MVC Helpers?

<b>A</b>	Inline HTML Helpers
<b>B</b>	Built-in HTML Helpers
<b>C</b>	Custom HTML Helpers
<b>D</b>	All of these

8. Which one of the following options is not an Action selector attribute?

<b>A</b>	ActionName
<b>B</b>	NonActionName
<b>C</b>	NonAction
<b>D</b>	ActionVerbs

9. Which of the following option is used to establish the identity of the user with the Internet?

<b>A</b>	No authentication
<b>B</b>	Individual user accounts
<b>C</b>	Work and school accounts
<b>D</b>	Windows authentication

10. Which of the following statements about partial views are true?

- a) Partial view is an exclusive view capable of rendering a portion of the view content.
- b) Partial views can be reused in multiple views.
- c) Code duplication can be reduced using partial view.
- d) The `ViewDictionary` object is used to instantiate the partial view.
- e) To render any related data in a partial view, which is part of the model, one should ensure to use the `RenderPartial` method.

<b>A</b>	a, b, c
<b>B</b>	a, c, e
<b>C</b>	a, b, c, e
<b>D</b>	b, c, d

## Answers

1	D
2	B
3	D
4	B
5	A
6	D
7	D
8	B
9	C
10	C

## **Try It Yourself**

1. You are tasked with developing a secure document management system using ASP.NET MVC. The system will allow users to create, upload, view, and manage documents while enforcing strict security measures to protect sensitive data.
  - i. Implement authentication and authorization:
    - Users must be able to register, log in, and log out securely.
    - Only authenticated users should have access to the document management system.
    - Different roles (Admin, Manager, and Employee) will have varying levels of access rights.
      - Admins can create, edit, and delete documents, as well as manage user roles.
      - Managers can upload and view documents but cannot delete them.
      - Employees can only view documents.
  - ii. Implement role-based authorization to restrict access to specific actions and views based on user roles. Ensure that only users with appropriate roles can perform certain operations (only Admins can delete documents).

# Session 6: Action Methods and Advanced Concepts in MVC

## Session Overview

This session describes action methods in MVC. It compares Web API and SOAP. It explains bundling and minification in MVC. It also explains the process for sharing data for ASP.NET Core and MVC.

## Objectives

In this session, students will learn to:

- ✓ Explain Action Methods in MVC
- ✓ Compare Web API and SOAP
- ✓ Describe Route Config Declaration
- ✓ Describe the process for Sharing data for ASP.NET Core MVC
- ✓ Explain Bundling and Minification in MVC
- ✓ Explain Areas in MVC

## 6.1 Action Methods in MVC

Action methods in ASP.NET process requests and produce responses to them. Usually, a response is generated in the form of ActionResult. Actions normally have direct mapping with user interactions. For instance, open a Web browser and enter a URL. Once the URL opens, click any link on that page and submit it. Whenever such interaction happens, it sends a request to the server that contains data used by the MVC framework to invoke an action method.

Developers can understand how the action method works by considering a sample URL: <http://localhost:7575/PersonDetail/Index>

In this URL, it can be noticed that `PersonDetail` is the controller and `Index` is the action method. When a user enters this URL in the browser, he/she is invoking the action method `Index`. Table 6.1 lists the action method elements.

URL	Controller	Action
/PersonDetail/Index	PersonDetail	Index

**Table 6.1: Action Method Elements**

In simple terms, the browser sends a request to the controller, which in turn, invokes the method. If the method cannot be located in the controller, then it displays an `HttpNotFound` exception.

### 6.1.1 Controller with Action Methods in ASP.NET MVC

Action methods always return different view results. In ASP.NET, a method with `ActionResult` is known as an action method. When a developer creates an MVC

application and a controller is auto-generated, the controller class will have one or more action methods.

Code Snippet 1 shows a simple action method.

### Code Snippet 1

```
using Demo.Models;
namespace Demo.Controllers {
public class MemberDetailsController : Controller {
//
// GET: MemberDetails
[HttpGet]
public ActionResult Index() {
...
return View();
}
}
```

#### 6.1.2 ASP.NET MVC ActionResult

ActionResult signifies the result of action methods or return types of action methods that are described in `System.Web.Mvc` namespace. ActionResult is an abstract class, which serves as a base class for various types of action results.

The return value of an action method is a view, which is derived from `ViewResult` base class. A custom ActionResult return type can be developed by creating a class that inherits from `ActionResult` abstract class.

Various types of ActionResult supported by ASP.NET MVC framework are listed in Table 6.2.

Action Result	Helper Method	Description
ViewResult	View	Displays a view as an online page.
PartialViewResult	PartialView	Displays a partial view, which describes a fraction of a view that can be extended inside another view.
RedirectToRouteResult	RedirectToAction or RedirectToRoute	Diverts to another action method.
RedirectResult	Redirect	Diverts to another action method by using its URL.
JavaScriptResult	JavaScript	Returns an executable script.
ContentResult	Content	Results in a user-defined content type.
HttpNotFoundResult	HttpNotFound	Shows that the requested resource was not found.

Action Result	Helper Method	Description
FileResult	File	Returns a binary result to write to the outcome.
FileStreamResult	Controller.File(Stream, String) or Controller.File Stream, String, String)	Communicates the binary content to the response through a stream.
FileContentResult	Controller.File(Byte[], String) or Controller.File (Byte[], String, String)	Communicates the contents of a binary file to the response.
EmptyResult	(None)	Displays a return value if the action method must return a null result (void).

**Table 6.2: ActionResult**

## 6.2 Comparison Between Web API and SOAP

There are two different types of communication protocols in Web services, as follows:

- Simple Object Access Protocol (SOAP)
- Representational State Transfer (REST)

While SOAP was used in Web service interfaces as a standard protocol for a long time, of late, it has been overshadowed by REST.

### 6.2.1 REST or RESTful Service

The term REST refers to the architectural concept that helps in developing simple Web services, which assists in accessing and modifying Web resources discovered through URI.

Roy Fielding introduced REST, which was later used across the industry. To recognize the resources on the server, REST requires each resource to have a unique URI. For instance: <http://www.website.com/student/101> to access the student information having id 101. It returns the output in formats, such as XML, HTML, and JSON.

REST describes a set of protocol using which data can be transmitted over a standardized interface (such as HTTP). For instance, to access a student's information, one must make an HTTP

call to a Website with the help of SOAP as shown in Code Snippet 2.

### Code Snippet 2

```
<soap:Body xmlns:em=
"http://www.yoursite.org/
student">
<em:GetStudent>
<em:StudentId>101</em:Student
Id>
</em:GetStudent>
</soap:Body>
```

Now, a SOAP message is returned by the service, as shown in Code Snippet 3.

### Code Snippet 3

```
<soap:Body  
xmlns:em="http://www.yoursite.  
org/ student">  
<em:GetStudentResponse>  
<em:StudentId>101</em:StudentId  
>  
<em:Name>Tom Alexa</em:Name>  
<em:Email>tom.alex@gmail.com</  
em:Email>  
<em:DateOfJoin>12-Jun-2017</em:  
DateOfJoin>  
</em:GetStudentResponse>  
</soap:Body>
```

The request to fetch student details is triggered through a POST request. The service then returns the HTTP 200 status code along with the details of the student in a SOAP envelope.

The HTTP 200 status is returned even if the student id cannot be located on the server. However, the service displays an error message using the SOAP envelope.

The best way to utilize HTTP is to use the GET method to access or get a resource from server. To do this, the server must return a HTTP 200 status only if it can accommodate the requested resource. REST uses HTTP protocol, which is specified with the right usage of HTTP verbs, such as GET, POST, PUT, and DELETE. Suitable HTTP status codes are used, for example:

- 200 (only if the resource runs successfully)
- 201 (if a new resource is created using POST)
- 404 (if the requested resource is not found)

The example of REST API is shown as request in Code Snippet 2.

Response in Code Snippet 3 can be changed as shown in Code Snippets 4 and 5, respectively.

### Code Snippet 4

```
GET /student/101 HTTP/1.1  
http://www.website.com  
Accept: application/vnd.student+  
Json
```

### Code Snippet 5

```
HTTP/1.1 200 OK  
Content-Type:  
application/vnd.api+json  
{  
  "data": [  
    {  
      "StudentId": "101",  
      "Name": "Tom Alexa",  
      "Email":  
        tom.alex@gmail.com,  
      "DateOfJoin": "12-Jun-  
2017"  
    }  
  ]  
}
```

Code Snippets 4 and 5 showing request and response respectively prove that REST services are quite light.

This is because of absence of bulk XML coding. Further, these snippets use apt HTTP status codes.

The response item links enable the REST client to find the present resource if it is made using POST along with the other associated resources. This is known as hypermedia, a concept that allows the client to go to the next resource. Thus, a REST client uses the links of the REST response to go to the next resource.

REST architecture follows the Hypermedia As The Engine Of Application State (HATEOAS) constraint. It also works with the client and negotiates the content to reply in a format that the client understands.

Therefore, with the help of HTTP, a REST service offers efficient Web services by using hypermedia to handle assorted client-server connection.

### **6.2.2 Creating RESTful Services in ASP.NET**

RESTful service can be created employing ASP.NET Web API framework. As in MVC framework, ASP.NET Web API also has action, action filters, controllers, model binding, routing, model validations, and so on.

Even though it is assembled in a manner as in ASP.NET MVC, ASP.NET Web API works as an independent framework. The entire framework is distributed as distinct NuGet bundles that do not rely on ASP.NET MVC framework. The Web API project is facilitated in Internet Information Services (IIS) or it can be published in a different procedure.

Web API is also known as HTTP API and not as REST API for several reasons. In the REST architecture, an API not having hypermedia cannot be completely RESTful. Although ASP.NET Web API framework permits creating REST API, it no longer pressurizes to build each service exposed as REST API. It utilizes the REST design's potential of using HTTP programming model to allow lightweight Web services to support many clients.

### **6.2.3 Differences Between SOAP and REST**

Table 6.3 lists some of the differences between REST and SOAP.

<b>REST</b>	<b>SOAP</b>
REST works using regular interfaces to access specific resources.	SOAP works through various interfaces.
REST reveals components of application ideas as services.	SOAP reveals components of application ideas as data.
REST accesses data.	SOAP executes operations through a structured set of messaging design.
REST offered an easy way to access Web services as compared to SOAP by using HTTP.	SOAP was designed by Microsoft and proved to be a popular protocol.

**Table 6.3: Differences Between REST and SOAP**

#### **6.2.4 Benefits of REST Over SOAP**

Following are benefits of REST when compared to SOAP:

REST supports various data formats, but SOAP supports only XML. The browser-based clients preferably pick REST. Web-based services, such as eBay, Google, Amazon, and Yahoo use REST as a preferred protocol.

When it comes to performance, REST has an edge over SOAP as it uses caching technique for storing information which cannot be edited. REST is a faster and easier technique as compared to SOAP.

Existing Websites can be used without adding refactor site infrastructure. This allows developers to attach more functionality and save their time by not designing a site from scratch. Despite so many benefits of REST over SOAP, there are still cases where SOAP is highly recommended.

#### **6.2.5 Benefits of SOAP Over REST**

Following are benefits of SOAP that makes it a preferred protocol when compared to REST:

- Although REST is the most commonly used protocol, sometimes SOAP takes over the functionality of REST. SOAP is applied when a more secure application is built specially when WS-Security is desired. Using SSL, it also supports identity verification, which provides data privacy. SOAP has an additional feature of retry to check for failed connection whereas, REST does not come with this function. The advantage of a retry function is that the client must fix it by retrying.
- SOAP is designed using HTTP protocol that makes it simpler to work with various firewalls and securities, such as proxies without editing the base protocols. ACID-compliant transactions are carried with the help of SOAP as it has preferable high transactional reliability, which is not present in REST.
- SOAP is also used when an application is designed with limited codes. Such codes require complex operations, subject, and topic to be retained at the application layer for better security, transactions, and other components. It is compatible with other technologies, such as WS-ReliableMessaging, WS-Coordination, and WS-Addressing.
- Organizations decide which protocol to use depending on clients they are supporting and as per their requirement in terms of security and flexibility. REST and JSON are being majorly used because they take less bandwidth and are easy to understand and use by developers.

### 6.2.6 Choosing Between REST and SOAP

If there is a situation where one must choose between REST and SOAP, the following aspects will help in making a decision as per the requirements:

- SOAP Web service performs a POST operation at all times. On the other hand, while working with REST, there is an option to select HTTP methods, such as POST, DELETE, GET, and PUT.
- SOAP provides better security and reliability for the applications.
- REST is easy to implement as it has lesser code, whereas, SOAP is helpful when an API that is ever changing and complex is published.

Thus, it can be concluded that REST and SOAP have completely different functionality and work on different situations. While Web-based operations go well with REST and SOAP remotely manages the objects. The base technology on which REST was designed is HTTP, which leverages its own basic operations such as GET, DELETE, and POST. This allows REST to rule over public API applications.

On the other hand, SOAP uses XML and has features that allow beginners to specify connections between users and providers. SOAP transfers the data related to objects using XML, which consumes more bandwidth as compared to REST.

### 6.3 Route Config Declaration

As defined earlier, routing can be defined as a pattern matching approach that tracks the requests and decides where to send it for further processing. It attempts to match the URL pattern of the request with that available in the Route table. An example of a route is defined as shown in Code Snippet 6.

#### Code Snippet 6

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

Routing works on a simple methodology - search, match, and forward. It first finds the match in the Route table for the URL that comes as incoming requests. Then, it directs the request to the suitable controller. It shows the 404 HTTP error when there is no match for the incoming request's URL.

It is important for route names to be unique in the entire application. In MVC, if routing is done with additional modifications, the *RouteConfig* files is required.

The default Route is changed and ActionResults are provided by the corresponding Route mentioned in *RouteConfig*. While creating a site, there are chances of not using the *RouteConfig.cs* file at all. It basically

comes into picture when different routes are required for use cases.

To understand when there may arise a requirement to modify it, consider an example of a site: /Blog/Post/1234/my-blog-post, which has a space dedicated to blogs and that requires some modification.

RouteConfig follows the general standards and is prefixed by the Area name, as shown:

```
/ {area}/{controller}/{action}/{id}
```

Code Snippet 7 shows an example of how to override this.

### Code Snippet 7

```
context.MapRoute(
    "Blog",
    "Blog/Post/{id}/{name}",
    new { action =
        "Index", controller="Post" },
    new { id = @"\d+" } );
```

The URL example for this can be ~/blog/posts/123/Food.

To get a clear understanding between *RouteConfig.cs* and *global.asax.cs*, consider the codes shown in Code Snippets 8 and 9.

Code Snippet 8 shows the default code present in *RouteConfig.cs*.

### Code Snippet 8

```
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 });
    }
}
```

Code Snippet 9 shows the code as in *global.asax.cs*, which calls *RegisterRoutes* in *RouteConfig*.

### Code Snippet 9

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

Using config classes such as *routeconfig.cs*, *WebApiConfig.cs*, *AuthConfig.cs*, *BundleConfig.cs*, or *FilterConfig.cs* is considered as the best approach.

## 6.4 Bundling and Minification in MVC

The performance of a Website plays an important role in the Website development process. Therefore, it is important to enhance the speed of the Website and at the same time use lesser server resources. To achieve this, ASP.NET version 4.5 onwards offers built-in features called bundling and minification.

Generally, when a user accesses a Website from any browser, multiple requests are sent to the server, such as CSS, JavaScript, and jQuery. This in turn causes multiple responses. To accommodate this, ASP.NET 4.8.1 version onwards has features called '*bundle and minify JavaScript and CSS files*'.

A **bundle** can be defined as a logically grouped set of files having a unique name. It is then loaded with only one HTTP request. Bundles can be created for both JavaScript and CSS files.

**Minification** is the process in which redundant whitespaces, line breaks, and comments are eliminated from the code thus, reducing its size. This approach improves loading time for code.

Therefore, bundling and minification serve as performance optimization procedures that help in decreasing number of requests to the server and enhancing load times. They also help in decreasing the requested assets' size, for example, JavaScript and CSS files.

As mentioned earlier, the minification technique, when applied to JavaScript or CSS files, helps in removing unwanted whitespaces and comments. Variable names are also reduced to single character.

Consider a JavaScript function shown in Code Snippet 10.

### Code Snippet 10

```
function myFunction(x) {  
    var text;  
    /* If x is Not a Number or less than one or greater than 10 */  
    if (isNaN(x) || x < 1 ||  
        x > 10) {  
        text = "Input not valid";  
    } else {  
        text = "Input OK";  
    }  
    document.getElementById("message").innerHTML = text;  
}
```

The optimized and minimized version of Code Snippet 10 is shown in Code Snippet 11.

## Code Snippet 11

```
function myFunction(n){  
var t;  
t=isNaN(n) || <1 || n>10 ? "Input not valid" : "Input OK",  
document.getElementById("message").innerHTML=t;  
}
```

Code Snippet 11 shows how unwanted whitespaces and comments are removed. The variable names are also compressed, which eventually reduces the size of the JavaScript file. Thus, bundling and minification affects the loading time of the page. The loading time reduces because the size of the file and the number of requests is decreased.

Different bundle classes supported by MVC 5 and higher versions in System.Web.Optimization namespace are as follows:

### ScriptBundle

These are bundles that helps in minification of one or more JavaScript files.

### StyleBundle

These are bundles that help in minification of single or multiple CSS style sheet files.

### DynamicFolderBundle

This is a bundle object created by ASP.NET from a folder having same type of files.

#### 6.4.1 ScriptBundle

ScriptBundle class in ASP.NET MVC API does the bundling and minification for JavaScript. Following are steps to bundle many JavaScript files in one HTTP request:

Step 1: From the **MVC** folder, go to **App\_Start\BundleConfig.cs** file  
By default, the **BundleConfig.cs** file is generated by MVC.

Step 2: Write the customized bundling code in the **BundleConfig.RegisterBundles()** method  
Instead of using the **BundleConfig** class, developers can create custom classes. However, it is a good practice to follow standard procedures.

Code Snippet 12 shows a part of the default **RegisterBundles()** method.

## Code Snippet 12

```
using System.Web;  
using System.Web.Optimization;  
public class BundleConfig  
{  
    public static void RegisterBundles(BundleCollection bundles)  
    {
```

```

// create a ScriptBundle object and specify bundle name
// (as virtual path) as parameter
ScriptBundle spBndl = new
    ScriptBundle("~/bundles/bootstrap");
//use Include() to add all script files
spBndl.Include(
    "~/Scripts/bootstrap.js",
    "~/Scripts/respond.js"
);
//Include the bundle in BundleCollection
bundles.Add(spBndl);
BundleTable.EnableOptimizations = true;
}
}

```

Code Snippet 12 creates a bundle having two JavaScript files. They are bootstrap.js and respond.js and they use ScriptBundle.

Following steps show how bundling works:

- 1 Generate an instance of ScriptBundle class by including the bundle name as a constructor parameter.
- 2 The bundle name serves as a virtual path and starts with the symbol '~/'. Anything can be given in the virtual path, but it is best to assign a name that can be easily identified as a bundle. In Code Snippet 3, the name given is '~/bundles/bootstrap'. Thus, it is evident that the bundle includes files associated with bootstrap.
- 3 With the help of Include method, add multiple 'JS' files into a bundle. This is achieved by following the root path with the relative path identified by the symbol "~".
- 4 Include the bundle into the BundleCollection instance. This is a parameter in the RegisterBundle() method.
- 5 Set **BundleTable.EnableOptimizations=True**. This allows for bundling and minification in the Debug mode. In case it is set to false, it will not do the bundling and minification functions.

Another way to bundle all the files would be to use the `IncludeDirectory` method. This method includes all the files under a particular directory as shown in Code Snippet 13.

### Code Snippet 13

```
public static void RegisterBundles(BundleCollection bdl) {
    bdl.Add(new
    ScriptBundle("~/bundles/JsScripts").IncludeDirectory("~/JsScripts"
    "/",
    "*.js", true));
}
```

During the start of an application, the MVC framework calls the `BundleConfig.RegisterBundle()` method from the event `Application_Start` in the `Global.asax.cs` file so that the bundles are packed into `BundleCollection` as shown in Code Snippet 14. This is the auto-generated code.

### Code Snippet 14

```
public class Global : HttpApplication{
protected void Application_Start() {
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
}
```

#### 6.4.2 Using Wildcards

The name of the script file can sometimes include versions of script files by third party. Therefore, when the version of the script file undergoes upgradation, it is not recommended to make changes in the code. This can be accomplished with the use of wildcards. When wildcards are used, appropriate versions of the files are automatically included.

To explain further, jQuery statement would indicate the version in a name. In which case, the `{version}` wildcard would pick up the available appropriate version of the file. Code Snippet 15 depicts such an implementation.

In Code Snippet 15, the version of the file associated with the jQuery statement will be included in the bundle. Thus, if `jquery-1.7.1.js` is the statement, it will include file version 1.7.1. Similarly, an advanced version of the file would be included by the statement `jquery -1.10.2.js` without making any changes in the code or compiling it.

Code Snippet 15 shows the auto-generated code.

### Code Snippet 15

```
public class BundleConfig {
    public static void RegisterBundles(BundleCollection bdl) {
        bdl.Add(new ScriptBundle("~/bundles/jquery")
            .Include( "~/Scripts/jquery-{version}.js"));
    }
}
```

#### 6.4.3 Using Content Delivery Network (CDN)

CDN also helps to load the script files. The `jquery` library can be loaded from the CDN. Code Snippet 16 depicts such an auto-generated implementation.

#### Code Snippet 16

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bdl) {
        var cdnPath = "http://ajax.aspnetcdn.com/ajax/jquery/jquery-
1.7.1.min.js";
        bdl.Add(new ScriptBundle("~/bundles/jquery", cdnPath)
            .Include("~/Scripts/jquery-{version}.js"));
    }
}
```

In Code Snippet 17, `jquery` will load files from the CDN when in the Release mode. However, when in the Debug mode, the `jquery` library gets loaded from the local source. Thus, it is important to have an alternative mechanism in case of a CDN request failure.

Following are the steps to include the `ScriptBundle` into a Razor view considering the `ScriptBundle` created in Code Snippet 17:

1. To get the bundles into Razor view, script bundles must be included with the help of static scripts class.
2. Use `Scripts.Render()` method to include the explicit script bundle during runtime. Code Snippet 17 depicts this auto-generated implementation.

#### Code Snippet 17

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
@Styles.Render("~/Content/bootstrap")
</head>
<body>
    /* HTML content will come here */
    @Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
</body>
</html>
```

When Code Snippet 17 is executed, two script files will be bundled, minified, and loaded as a single request. The Debug mode should be set to **OFF** as shown:

```
web.config<compilation debug="false" targetFramework="5" />
```

#### 6.4.4 StyleBundle

StyleBundle procedures are similar to ScriptBundle. The method of adding a virtual path and file names/patterns in ScriptBundle is followed in StyleBundle as well.

Code Snippet 18 depicts the default built-in code used for configuring the `BundleConfig.cs` file.

#### Code Snippet 18

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bdl)
    {
        bdl.Add(new ScriptBundle("~/bundles/jquery").Include(
                  "~/Scripts/jquery-{version}.js"));
        bdl.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                  "~/Scripts/jquery.validate*"));
        // Use the Modernizr's development version. Once ready for
        // production, consider using the build tool as per the
        // required tests at http://modernizr.com.
        bdl.Add(new ScriptBundle("~/bundles/modernizr").Include(
                  "~/Scripts/modernizr-*"));
        bdl.Add(new ScriptBundle("~/bundles/bootstrap").Include(
                  "~/Scripts/bootstrap.js",
                  "~/Scripts/respond.js"));
        bdl.Add(new StyleBundle("~/Content/css").Include(
                  "~/Content/site.css"));
        bdl.Add(new StyleBundle("~/Content/bootstrap").Include(
                  "~/Content/bootstrap.css"));
    }
}
```

Code Snippet 19 depicts an implementation in which registration of bundles is done in `Global.asax` file within the `Application_Start()` method.

#### Code Snippet 19

```
using System;
using System.Web;
using System.Web.Routing;
using System.Web.Optimization;
namespace StudentManagement{
```

```
public class Global : HttpApplication {
    void Application_Start(object sender, EventArgs e)
    {
        // Code to run on startup
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

Code Snippet 20 depicts an implementation where the bundles can be seen in various views.

### Code Snippet 20

```
@using System.Web.Optimization
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/modernizr")
@Styles.Render("~/Content/CSS")
@Styles.Render("~/Content/themes/base/CSS")
```

It is important to note that bundling cannot be done in the development mode of the application. This is because the compilation element debug is set to 'true' (debug="true") in the Web.config file. In the debug mode, the render statements in the views will have files in non-bundle and non-minified format.

In the production mode, the compilation element debug is set to 'false' (debug="false"). This would result in bundling of files.

This could cause confusion for scripts that use the reference relative paths existing in other files. An example of this is references to Twitter Bootstrap's icon files. The issue can be resolved by using the procedure, System.Web.Optimization's CSSRewriteUrlTransform class as shown in Code Snippet 21.

### Code Snippet 21

```
bundles.Add(new
    StyleBundle("~/bundles/css").Include("~/Content/css/*.css", new
    CSSRewriteUrlTransform()));
```

The CSSRewriteUrlTransform class replaces the relative URLs in the bundled files to the absolute paths. This ensures that the references are retained even if the calling reference shifts to the bundle's location.

For example, **~/Content/CSS/bootstrap.CSS** can be moved to **~/bundles/CSS/bootstrap.CSS** by applying the implementation given in Code Snippet 21.

#### 6.4.5 Minification

Minification is a procedure used to decrease the size of CSS and JavaScript files, thus reducing the time required for downloading. This is achieved by discarding unwanted whitespaces, comments, and other unimportant content from the files.

This procedure involves the usage of `ScriptBundle` or `StyleBundle` object in which case it takes place automatically. For disabling this feature, the basic `Bundle` object can be used.

Code Snippet 22 depicts an implementation where pre-processor directives are used to perform bundling only during releases. This makes debugging easy in case of non-release runs as navigation through files that are not bundled is faster.

#### Code Snippet 22

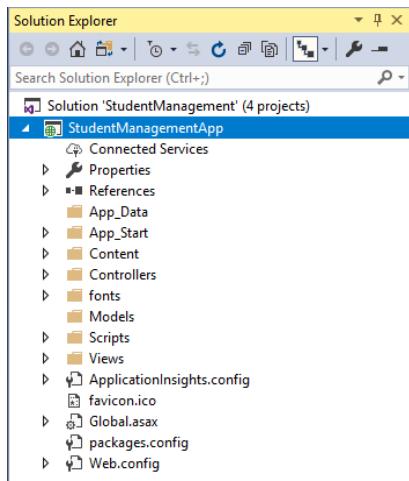
```
public static void RegisterBundles(BundleCollection bdl)
{
    #if DEBUG
        bdl.Add(new
    Bundle("~/bundles/jquery").Include("~/Scripts/jquery-
{version}.js"));
        bdl.Add(new
    Bundle("~/Content/CSS").Include("~/Content/site.CSS"));
    #else
        bdl.Add(new
    ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-
{version}.js"));
        bdl.Add(new
    StyleBundle("~/Content/CSS").Include("~/Content/site.CSS"));
    #endif
}
```

#### 6.5 Areas in ASP.NET MVC

The ASP.NET MVC project template has the provision to separate the application code depending upon the responsibilities of the people handling it. When a new ASP.NET MVC application is created, following folders are created in the root directory:

- Controllers
- Views
- Models

Figure 6.1 shows the project folder structure.



**Figure 6.1: Project Folder Structure**

On the top level, there are folders called **Controllers**, **Views**, and **Models**. Under the **Controllers** folder, there are multiple controllers, catering to a group of specific functionalities. Under the **Views** folder, there are multiple folders associated with a controller bearing the same name. This consists of View files meant for that specific controller. The Action names in the controller are mapped onto the View files (.cshtml file).

This classification works fine as long as the number of controllers and views in the application are few in number. As their number increases, it becomes necessary to organize and manage the code efficiently. Use of areas comes in handy for organizing the code in this scenario. **Areas** are logical groupings of Controllers, Views, Models, and all related files to a module in the MVC application. Typically, an Area folder at the top level consists of multiple areas. The areas assist in writing better maintainable code for an application by segregating it as per the modules.

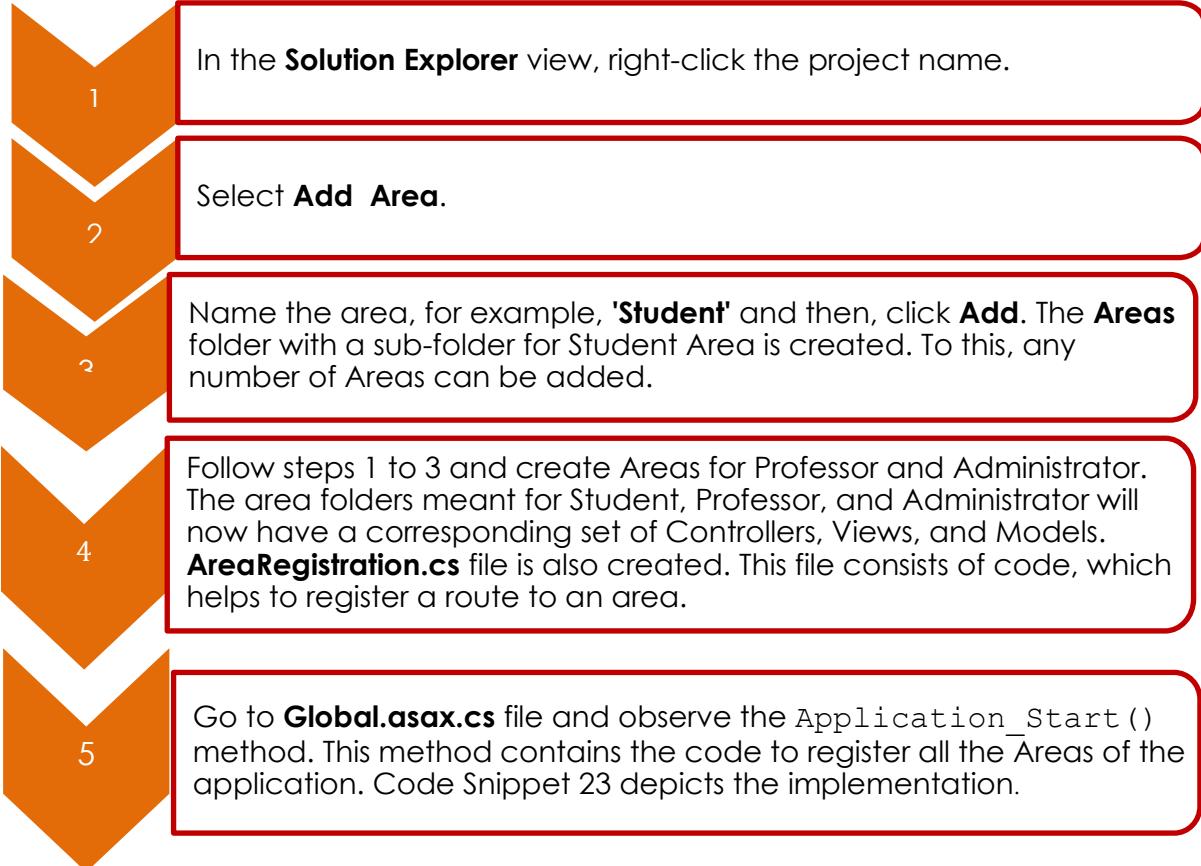
Each area consists of following parts:



To understand the concept better, consider an example of building a student portal. Typically, a student portal consists of the following functional areas:

Student Area	Professor Area	Administrator Area
In this functional area, a student can create his or her profile, check, and upload assessments	In this functional area a professor can generate and upload assignments, and also check student's performance	In this functional area, an administrator is given control to configure and manage the application

The procedure to create an area in the MVC application is as follows:

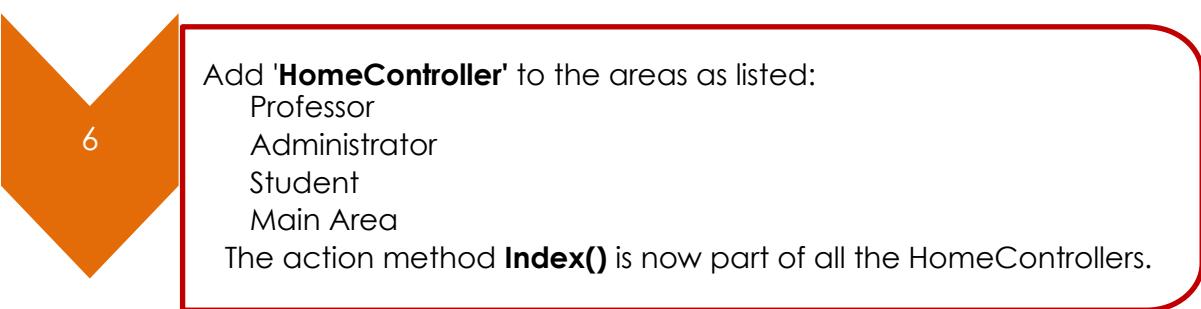


### Code Snippet 23

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);

    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
}
```



Code Snippet 24 depicts the auto-generated code.

## Code Snippet 24

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

7

To all the Areas, insert 'Index' views by copying and pasting the following in their respective views:

**Main Area:** <h1>Main Area Index View</h1>

**Student Area:** <h1>Student Area Index View</h1>

**Professor Area:** <h1>Professor Area Index View</h1>

**Admin Area:** <h1>Admin Area Index View</h1>

8

Create the application and go to **/MVCDemo**. At this point, an error is displayed. To rectify the error, include `RegisterRoutes()` method in the **RouteConfig.cs** file available in the **App\_Start** folder.

Code Snippet 25 shows how to route the namespace in the `HomeController` to the Main area using the namespace parameter.

## Code Snippet 25

```
public static void RegisterRoutes(RouteCollection rts) {
    rts.IgnoreRoute("{resource}.axd/{*pathInfo}");
    rts.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
id = UrlParameter.Optional },
        namespaces: new [] { "MVCDemo.Controllers" }
    );
}
```

9

Now, **/MVCDemo/Student** displays an error which states "**Resource cannot be found**".

To rectify the error, **RegisterArea()** area method in **StudentAreaRegistration.cs** file under the Student folder must be modified as shown in Code Snippet 26.

## Code Snippet 26

```
public class StudentAreaRegistration : AreaRegistration {
    public override string AreaName     {
        get { return "Student"; }
    }
    public override void RegisterArea(AreaRegistrationContext
        context)      {
        context.MapRoute(
            "Student_default",
            "Student/{controller}/{action}/{id}",
            new { action = "Index", id =
                UrlParameter.Optional })
    }
}
```

ActionLink() HTML helper could be used to move around different areas. Once the area name is specified, one can move from one area to another by selecting that particular link.

10

Open <http://location/MVCDemoStudent>, which could result in a compilation error linked to System.Web.Optimization. To rectify the error, perform the following:  
In Visual Studio 2022, click **Tools Library Package Manager Package Manager Console**.  
In the window that appears, enter the command: **Install-Package Microsoft.Web.Optimization -Pre** and press **Enter**.

## 6.6 Introduction to Data Sharing

Any data that is sent from a Web application to the server is considered important. Data is sent from one page of a Website to other, where actual business logic layer performs various activities. Performing these tasks would be rather difficult for a developer.

For example, writing code to store data values first on the server and then, saving or even deleting data from the server once it has been used. To accomplish these tasks, ASP.NET offers various methods that the developer can use.

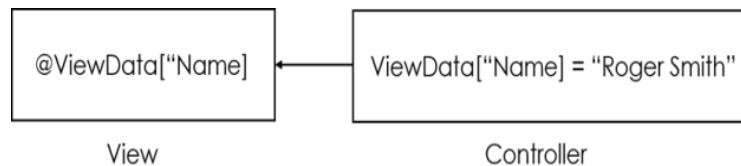
One of these is the action method in a controller that processes user requests and produces results. These results are presented in the form of a view to the user. ASP.NET provides following data sharing techniques to transfer data from a controller to view:

- ViewData
- ViewBag
- TempData
- Strongly typed view

### 6.6.1 ViewData

**ViewData** is called as a KeyValue pair object or a Dictionary object. Using ViewData, developers can transfer data from a controller to a view. By default, the type of ViewData is an object of ViewDataDictionary class. In this Dictionary object, Strings are used as keys to store and access value or data from ViewData.

Figure 6.2 displays how the data is passed from the controller to view in ViewData.



**Figure 6.2: ViewData**

Assume that an MVC application StudentManagement is created with a Model named Course. Code Snippet 27 represents how to use ViewData to transfer data from the controller to view. In the code, a CourseList with key 'Course' is added in ViewData dictionary.

#### Code Snippet 27

```
public ActionResult Index() {
    IList<Course> CourseList = new List<Course>();
    CourseList.Add(new Course() {
        CourseName = "Working with ASP.NET" });
    CourseList.Add(new Course() {
        CourseName = "Working with Azure" });
    CourseList.Add(new Course() {
        CourseName = "Working with Python" });
    ViewData["Courses"] = CourseList;
    return View();
}
```

In Razor view, this list can be accessed as shown in Code Snippet 28.

#### Code Snippet 28

```
@model StudentManagement.Models.Course
@using StudentManagement.Models
<ul>
@foreach (var std in ViewData["Courses"] as IList<Course>) {
    <li>@std.CourseName
    </li>
}
</ul>
```

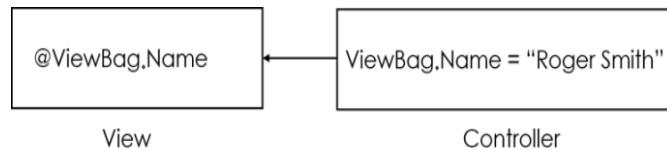
The code shows how to access ViewData in Razor view. In ViewData, the data assigned to value part is type sensitive and type casting is required. Code Snippet 29 displays how to add a KeyValuePair into ViewData.

### Code Snippet 29

```
public ActionResult Index() {
    ViewData.Add("CourseId", 1);
    ViewData.Add(new KeyValuePair<string, object>("CourseName",
    "Working with Python"));
    ViewData.Add(new KeyValuePair<string, object>("Duration",
    20));
    return View();
}
```

#### 6.6.2 ViewBag

Developers can use ViewBag to transfer data from the controller to the view. In case some data is not included in the model, ViewBag or ViewData. ViewBag is a dynamic data type property of the base class of all the controllers, which is the ControllerBase class. Figure 6.3 displays how temporary data is transferred from the controller to the view in ViewBag.



**Figure 6.3: ViewBag**

In Figure 6.3, the Name property is attached to ViewBag with the dot notation. A string value Roger Smith is assigned to it in the controller.

Developers can access it in the view, for example @ViewBag.Name (as per Razor syntax, @ is used when the server-side variable must be accessed).

Multiple properties and values can be assigned to ViewBag. However, when assigning multiple values to the same property, only the last value assigned will be stored. Data stored into ViewBag is stateless and it can be transferred only from the controller to the view. It cannot be transferred from one view to another view or from one controller to another controller. As ViewBag is stateless, in case of a redirection, the ViewBag values are set to null.

Assume that an MVC application is created with a Model named Employee. Code Snippet 30 displays how to set ViewBag in action method.

## Code Snippet 30

```
IList<Employee> employeeList;
public HomeController()
{
    employeeList = new List<Employee>() {
        new Employee(){ EmployeeID= 1, EmployeeName="Steve
                      Jones", Age = 21 },
        new Employee (){ EmployeeID=2, EmployeeName ="Bill Kin",
                      Age = 25 },
        new Employee (){ EmployeeID=3, EmployeeName ="Carrie
                      Stan", Age = 20 },
        new Employee (){ EmployeeID=4, EmployeeName ="Ron
                      Dixon", Age = 31 },
        new Employee (){ EmployeeID=5, EmployeeName ="Rob
                      Tucker", Age = 19 }
    };
}
public ActionResult Index()
{
    ViewBag.TotalEmployees = employeeList.Count();
    return View();
}
```

Here, `employeeList.Count()` will get Employee count and assign it to a `ViewBag` property named `ViewBag.TotalEmployees`. This can then be accessed in the `Index.cshtml` view to display the Employee info as shown in Code Snippet 31.

## Code Snippet 31

```
<label>Total Employees:</label> @ViewBag.TotalEmployees
```

The output of this code is total number of employees, which is 5. While retrieving values from `ViewBag`, typecasting is not required. A `ViewBag` is considered as a wrapper around `viewData`. If the `ViewBag` property name matches the key of `viewData`, a runtime exception error is displayed.

Both `ViewData` and `ViewBag` use `Dictionary` object internally and share a common memory. So, the same name cannot be used as `ViewBag` property name and `ViewData` Key simultaneously. This can cause a runtime exception.

Code Snippet 32 displays `ViewBag` and `ViewData`. As an item with that key has already been added earlier, a runtime exception error is displayed.

## Code Snippet 32

```
public ActionResult Index()
{
    ViewBag.CourseId = 1;
    ViewData.Add("CourseId",
```

```

    1); // throws runtime
    // exception as it already has // "Id" key
    ViewData.Add(new
        KeyValuePair<string,
        object>("CourseName",
        "Working with Azure"));
    ViewData.Add(new
        KeyValuePair<string,
        object>("Duration", 20));
    return View();
}

```

The stateless property of ViewBag can store data only in the current HTTP request. For every server request, the ViewBag is updated with the new value or is reset to null.

Following are some of the main aspects of ViewBag:

- ViewBag can be used to transfer data which is not in the model from the controller to view.
- ViewBag is a dynamic data type, which internally uses ViewData to store values.
- Use ViewBag to store multiple properties and values.

### 6.6.3 TempData

TempData is used only for current and subsequent requests as it is a very short-lived instance. Due to this reason, it is important to know what the next request is, which is assured only when it is redirected to another view. Thus, redirecting is the only case when users can rely on TempData. This is because when redirecting, current request is killed, and a new request is created on the server to serve the redirected view.

Sharing data between the controller actions are done through the ASP.NET MVC **TempData** dictionary. TempData value lasts until it is read or until the session times out of the current user's session. However, all the content is saved to the session state by TempData.

While reading, TempData values are marked for deletion. All marked values are then deleted at the end of the request. The advantage of this feature is that TempData will not be empty if there is a chain of multiple redirections. The values are retained until they are used and thereafter, automatically clear themselves.

Code Snippet 33 displays how to assign a value to TempData in a controller named WebController.

#### Code Snippet 33

```

public class WebController : Controller{
    public ActionResult Index()
    {

```

```
 TempData["tdata"] =  
 "TemporaryInfo";  
 return View();  
}  
}
```

Perform these steps to create a view to show this data:

1. Right-click the `Index` method in the controller class.
2. Select **Add View**.
3. Specify a suitable name, such as `Index.cshtml`. Open the view file and write the code shown in Code Snippet 34 to display data on the Website.

#### Code Snippet 34

```
@{  
    ViewBag.Title = "Index";  
}  
<h2>Index</h2>  
@TempData["tdata"]
```

4. Execute the code to check if output is as expected.

**Note:** Use `Keep()` to retain `TempData` values for the next request even after reading them. For example, `TempData.Keep("test")`

Use `Remove()` to remove `TempData` values from the current and next request. For example, `TempData.Remove("test")`

#### 6.6.4 Strongly Typed View

Strongly typed view is the view that bonds any model with a view. Any class can be bound as a model to the view. Model properties can be accessed on this view. Any data associated with the model can be used to render controls. Custom types and primitive types, such as string, array, list, and so on can be bound to the view.

Code Snippet 35 represents an object created from `StudentCourseModel` and it is passed to a strongly typed view.

#### Code Snippet 35

```
public ActionResult StudentCourse(){  
    StudentCourseModel courseModel = new StudentCourseModel();  
    return View(courseModel)  
}
```

Code Snippet 36 displays a strongly typed view.

#### Code Snippet 36

```
@Html.DisplayFor(modelItem => item.CourseId)
```

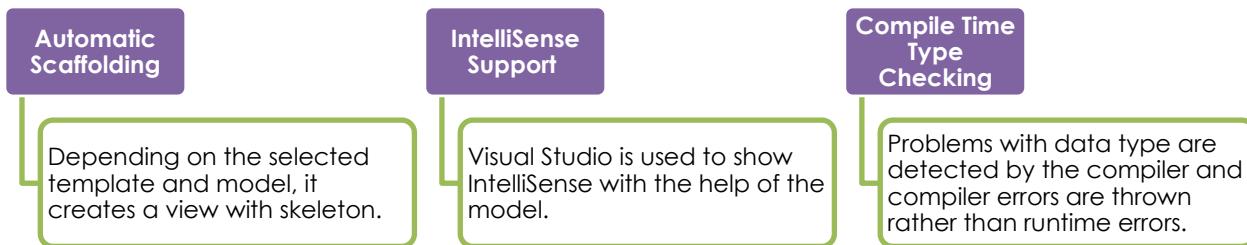
In strongly typed view, while posting the view, the same object can be used. Hence, the values assigned to the controls will be associated with the model's property.

Code Snippet 37 displays how the post action helps to access the same object.

### Code Snippet 37

```
[HTTPPOST] public ActionResult View1(UserModel obj) {  
    obj.save();  
    return view();  
}
```

To post the model at the server-side, use strongly typed view. For example, if a developer uses strongly typed view to add something, then more details can be added to post action. The developer can also save the operation on that object. Some important features of strongly typed view are as follows:



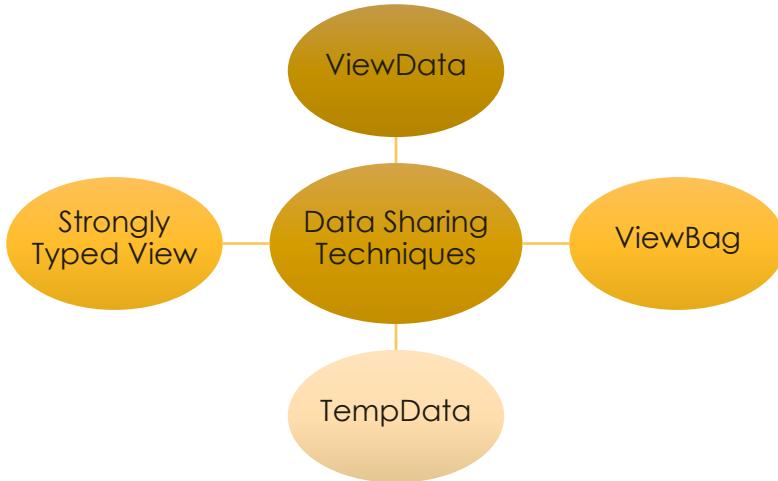
Following are various advantages displayed by strongly typed view over standard view:

- It retrieves values from `ViewData.Model`, rather than setting them in properties.
- It supports IntelliSense and type safety.
- It does not include any unnecessary casting between types in `ViewData`.
- It implements compile-time checks.

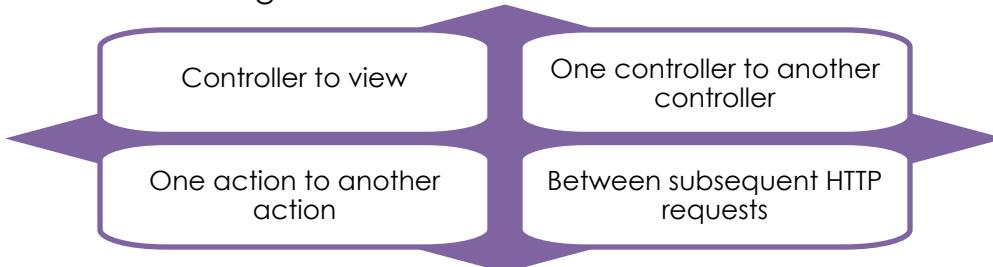
## 6.7 Data Sharing Techniques

While requests are processed and results are prepared by the action methods in the controller, it is the view that helps to present these results to the user.

Following methods are used to transfer the results from the controller to the view:



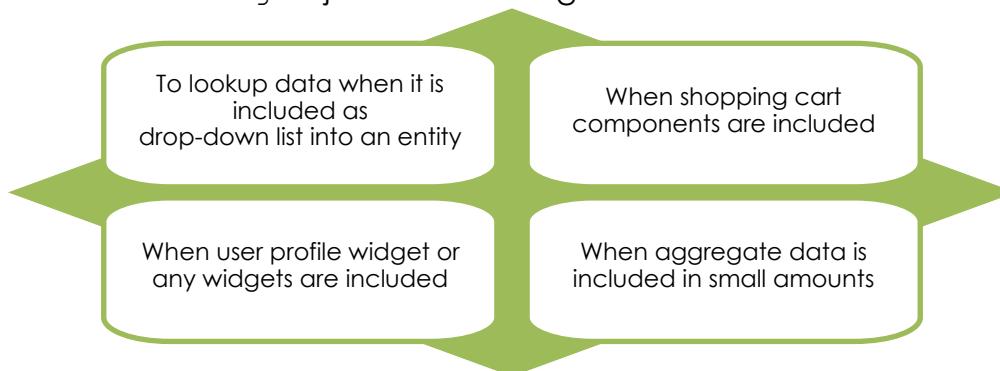
**ViewData**, **ViewBag**, and **TempData** are ASP.NET MVC objects. They help to carry or transfer data in the following scenarios:



#### 6.7.1 Comparing Data Sharing Techniques

In short, **ViewBag** is a dynamic object that helps to transfer data from the controller to view. **ViewData** is a dictionary object that helps to transfer data from the controller to view. While, **TempData** is a dictionary object that helps to transfer data from one controller/action to another controller/action. **ViewData**, **ViewBag**, and **TempData** include properties of both the view and controller. To carry any amount of data from and to specific locations (for example, controller to view or between views), use **ViewData**, **ViewBag**, and **TempData**.

Use **ViewData** and **ViewBag** objects in following scenarios:



`TempData` object is used to carry data between the current and next HTTP requests. Heavy duty `ViewModel` object is used while handling larger amounts of data, while reporting data, creating dashboards, or while working with multiple disparate sources of data.

While `ViewData` and `ViewBag` are suitable for transferring small amount of data between the controller and view, they may become large to manage for larger or more complex data sets.

`TempData`, on the other hand, is specifically designed for short-term storage of data between sequential requests and is suitable for scenarios where data must persist across actions or redirects within a single user session.

Table 6.3 displays the comparison between `ViewData`, `ViewBag`, and `TempData`.

<b> ViewData</b>	<b> ViewBag</b>	<b> TempData</b>
It transfers data from the controller to the view. It can be accessed using strings as keys.	A dynamic wrapper around <code>ViewData</code> is <code>ViewBag</code> .	If data is required for the next request also, then use <code>TempData</code> . Note that the data will be gone after the next request.
It obtains a null value in case of redirection. However, it also requires typecasting for complex data types.	It is found only in ASP.NET MVC 3 onwards. It can handle complex data types without typecasting. It obtains a null value in case of a redirection.	Data is passed from the current request to the subsequent request (redirection) using <code>TempData</code> . Due to this reason, <code>TempData</code> value will not be null.

**Table 6.3: Comparison Between `ViewData`, `ViewBag`, and `TempData`**

# Summary

- ✓ In an MVC application, a controller is auto-generated and will have one or more action methods.
- ✓ There are two different types of communication protocols in Web services, SOAP and REST.
- ✓ REST supports various data formats, but SOAP supports only XML. Browser-based clients preferably pick REST.
- ✓ REST uses HTTP protocol, which is specified with the right usage of HTTP verbs, such as GET, POST, PUT, and DELETE. SOAP is designed using HTTP protocol that makes it simpler to work with various firewalls and securities.
- ✓ Routing can be defined as a pattern-matching approach that tracks requests and decides where to send it for further processing.
- ✓ A bundle can be defined as a logically grouped set of files having a unique name. It is then loaded with only one HTTP request.
- ✓ Minification is the process in which redundant whitespaces, line breaks, and comments are eliminated from the code thus, reducing its size.
- ✓ ScriptBundle class in ASP.NET MVC API does the bundling and minification for JavaScript.
- ✓ Content Delivery Network (CDN) helps to load script files. The `jquery` library can be loaded from the CDN.
- ✓ Areas are logical groupings of Controllers, Views, Models, and all related files to a module in the MVC application.
- ✓ ViewData is called as a KeyValue pair object or a Dictionary object. Using ViewData, developers can transfer data from a controller to a view.

# Test Your Knowledge



1. Which of the following Helper method in ActionResult returns a user-defined content type?

<b>A</b>	Content
<b>B</b>	RedirectToAction
<b>C</b>	Redirect
<b>D</b>	PartialView

2. Which of the following ActionResult communicates the file contents to the response?

<b>A</b>	EmptyResult
<b>B</b>	FilePathResult
<b>C</b>	FileResult
<b>D</b>	ContentResult

3. Which of the following has the extension '\*.cshtml'?

<b>A</b>	Model
<b>B</b>	View
<b>C</b>	Controller
<b>D</b>	None of these

4. A \_\_\_\_\_ can be defined as a .NET component that can attach itself to the application lifecycle and offer functionality.

<b>A</b>	Action method
<b>B</b>	View
<b>C</b>	Module
<b>D</b>	Controller

5. Which of the following is the process of generating .NET objects with the help of the data transmitted by the browser in an HTTP request?

<b>A</b>	Action method
<b>B</b>	Business logic
<b>C</b>	Model binding
<b>D</b>	None of these

6. Which of the following format is not returned while using REST?

<b>A</b>	XML
<b>B</b>	HTML
<b>C</b>	GET
<b>D</b>	JSON

7. Which of the following is the best way to utilize HTTP method?

<b>A</b>	POST
<b>B</b>	DELETE
<b>C</b>	GET
<b>D</b>	RESULT

8. For which of the following the data associated with the model can be used to render controls?

<b>A</b>	ViewData
<b>B</b>	ViewBag
<b>C</b>	TempData
<b>D</b>	Strongly typed view

## Answers

1	A
2	B
3	B
4	C
5	C
6	C
7	C
8	D

## Try It Yourself

1. Create an ASP.NET MVC application with a controller and view to display a simple message.
2. Extend the previous application to include a form for user input and handle form submission and optimize the application's performance by implementing bundling and minification for CSS and JavaScript files.
3. Create a model class named `Book` with properties for `Id`, `Title`, `Author`, `Genre`, and `Price`. Implement Create, Read, Update, Delete (CRUD) operations for managing books. Use Entity Framework Code First approach for data access and database management.

Create views and corresponding action methods in the `BooksController` to:

- a. Display a list of all books.
  - b. View details of a specific book.
  - c. Add a new book.
  - d. Edit an existing book.
  - e. Delete a book.
- 
4. Create an ASP.NET MVC application for a simple blogging platform that allows users to create, view, update, and delete blog posts. Design a database schema with two tables: Posts and Comments. The Posts table should have columns for `Id`, `Title`, `Content`, `Author`, and `DateCreated`. The Comments table should have columns for `Id`, `PostId` (foreign key to Posts table), `Author`, `Content`, and `DateCreated`.

Implement CRUD operations for managing blog posts and comments. Use Entity Framework Code First approach for database management.

Create views and corresponding action methods in the `PostsController` to:

- a. Display a list of all blog posts.
- b. View details of a specific blog post along with its comments.
- c. Add a new blog post.
- d. Edit an existing blog post.
- e. Delete a blog post.
- f. Add a new comment to a blog post.

# *Session 7: Enhancements in ASP.NET Core*

## **Session Overview**

This session describes .NET Core MVC and Razor pages. It explains MVC Model Binding. It also describes tool support for dump debugging. Finally, it describes Garbage Collection.

## **Objectives**

In this session, students will learn to:

- ✓ Describe .NET Core MVC and Razor page enhancements.
- ✓ Explain MVC Model Binding enhancements
- ✓ Describe Text.Json
- ✓ Describe dump debugging
- ✓ Outline Performance improvements in recent versions
- ✓ Identify the use of Garbage Collection

## **7.1 .NET Core MVC and Razor Pages**

Razor Pages was first introduced in ASP.NET Core MVC 3.0. It is an engine used to create views of MVC design pattern in Microsoft Visual Studio. Razor uses a markup syntax to embed server-based code in languages such as Visual Basic and C# into Web pages.

A view-engine is a pluggable module that can implement different template syntax options and is designed to improve productivity.

In general, while dynamic Web content can be created rapidly using server-based code, a Web page is to be created for the browser.

When a request for a Web page is made by an end-user, the server-based code is executed by the server inside the page. This is performed before the page is returned to the browser. The code performs complex tasks, such as accessing databases when the server is running.

Razor is designed in such a way that it helps to easily create such Web applications. Using Razor, developers can start with static HTML (or any textual content) and make it dynamic by adding server code to it.

Although it is traditional ASP.NET markup, it is easy to use and learn Razor. The functionality of Razor is similar to Active Server Page eXtended (ASPX) files.

ASP.NET MVC and Core MVC support the Razor view engine. Both HTML and server-side code using C# or Visual Basic (VB) are employed for writing Razor syntax. In Visual Studio 2019 onwards, with ASP.NET MVC, Razor view engine is built-in by default.

Following are the file extensions in the Razor view:

- .vbhtml file extension for VB syntax
- .cshtml file extension for C# syntax

Some characteristics of Razor syntax are as follows:

**Compact**

Enables to write easy and simple codes by minimizing the number of characters and keystrokes

**Easy to Learn**

Enables use of languages such as C# or VB

**IntelliSense**

Includes support within Visual Studio in Razor syntax for statement completion

Rules of Razor syntax for C# are as follows:

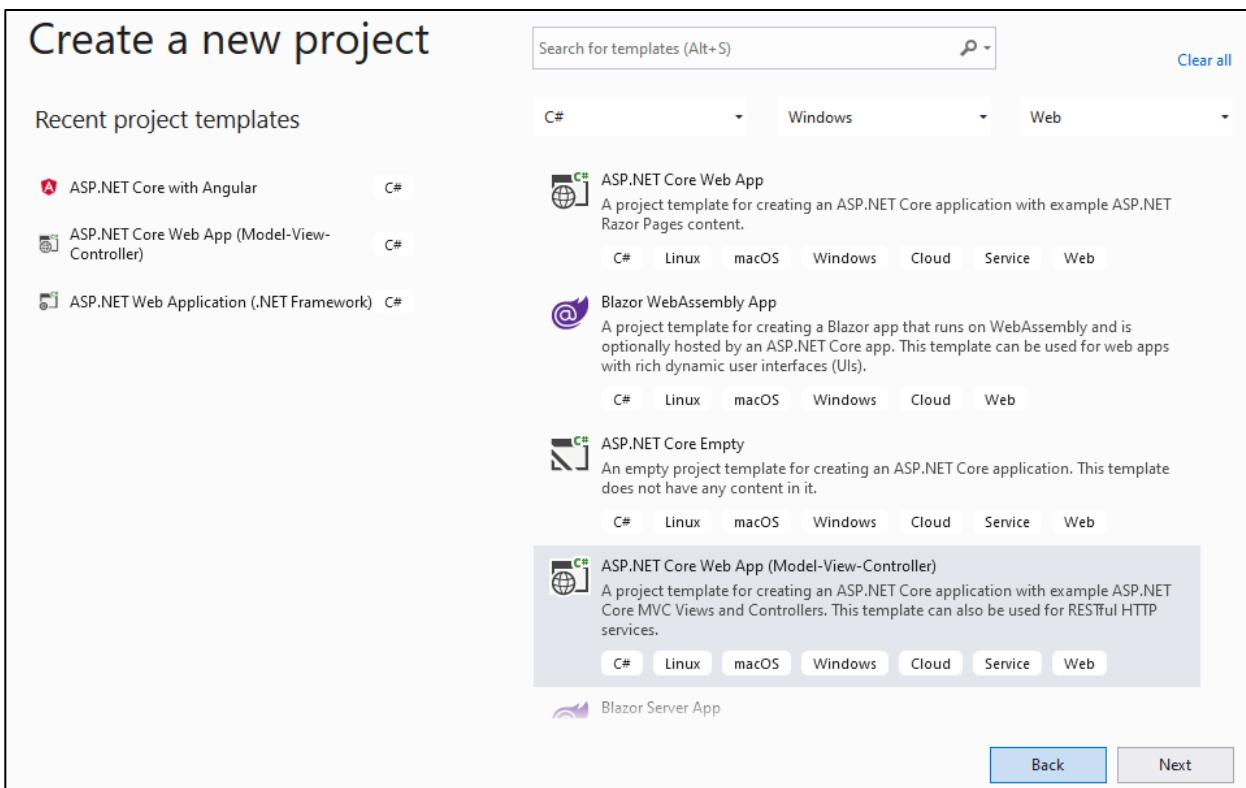
- Use @{ ... } for Razor code blocks.
- Start inline expressions (variables and functions) with @.
- Use semicolon for ending code statements.
- Use var keyword to declare variables.
- Use quotation marks to enclose strings.
- Adhere to case sensitivity of C# code.

Use .cshtml extension for C# files.

### 7.1.1 Razor-based MVC Application

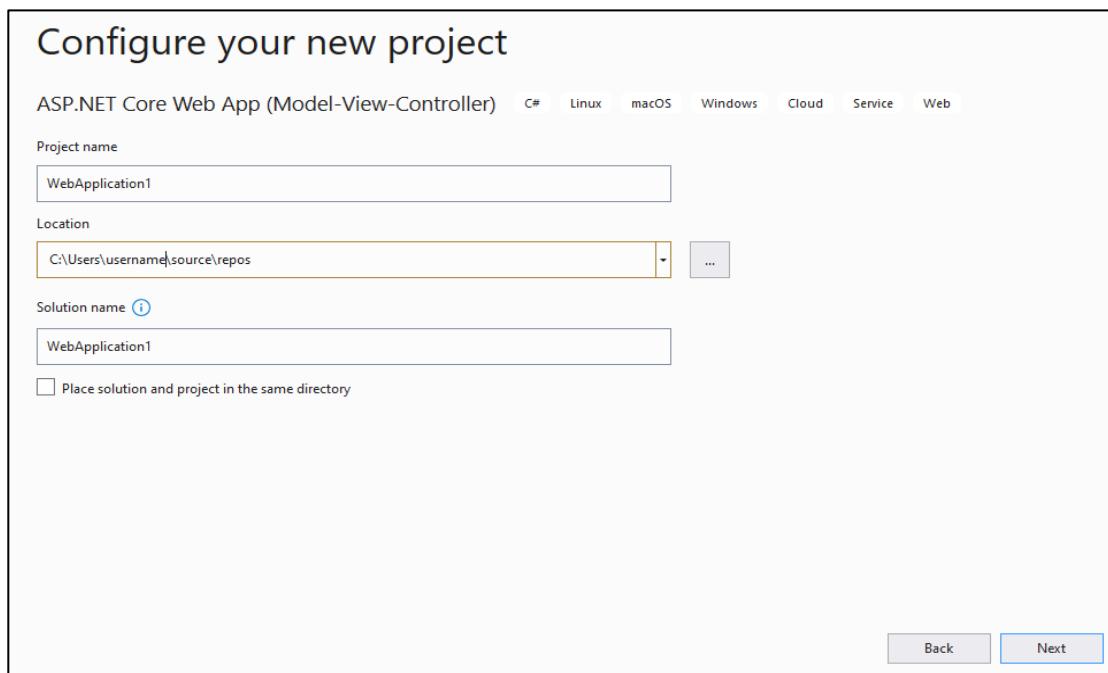
Following steps are performed to create a Razor based MVC application:

In Visual Studio 2022, start by creating a new project. In the **Create a new Project** window, select **ASP.NET Core Web App** as shown in Figure 7.1.



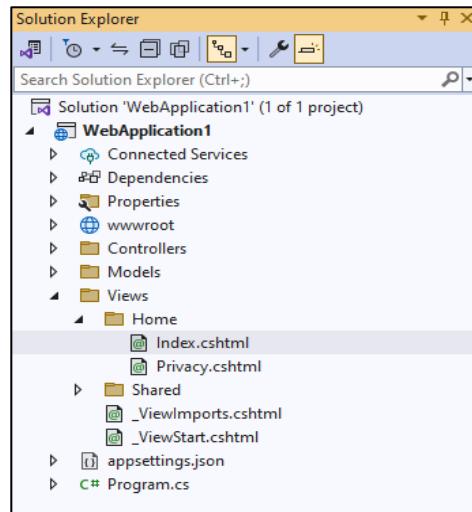
**Figure 7.1: Creating a New Project**

1. In the next window, add a name for the project, specify the location, and click **Next** as shown in Figure 7.2.



**Figure 7.2: Configure New Project**

The application is created with default folder structure as shown in Figure 7.3.



**Figure 7.3: Directory Structure in Solution Explorer**

HTML content and Razor code are two types of content that can be observed on a Razor Web page. For a server to read a page, the Razor code is run first. It then sends the HTML page to the browser. Tasks that cannot be performed in the browser are performed by the server after executing that code. For example, accessing a server database.

Server-side C# or VB code is written along with the HTML code starting with @ symbol.

Some examples are as follows:

- To display a value of a server-side variable, write @Variable\_Name
- To display current DateTime, write @DateTime.Now

A semicolon is not required at the end of a single line expression.

Code Snippet 1 displays a simple example of how to write server-side C#. Consider that a page `StudentInfo.cshtml` is created in which this code is added.

### Code Snippet 1

```
@page
<h1>Razor demo</h1>
<h2>
@DateTime.Now.ToShortDateString()
</h2>
```

### Output:

Razor demo  
28-02-2024

Multiple lines of server code can be written in braces @{ ... }. Similar to C#, in this case, each line ends with a semicolon.

Code Snippet 2 displays an example of the same.

## Code Snippet 2

```
@page
#{@
var date =
DateTime.Now.ToString("dd-MM-yyyy");
var message = "I am nervous!";
}
<h2>My exam is on: @date </h2>
<h3>@message</h3>
```

### Output:

My exam is on: 28-02-2024  
I am nervous!

Text within a code block is displayed using @: or <text>/<text> as shown in Code Snippet 3A and Code Snippet 3B, respectively.

## Code Snippet 3A

```
@{
    var date = DateTime.Now.ToString("dd-MM-yyyy");
    string message = "Hello All!";
    @:Today's date is: @date <br />
    @message
}
```

## Code Snippet 3B

```
@{
    var date = DateTime.Now.ToString("dd-MM-yyyy");
    string message = "Hello All!";
    <text>Today's date is:</text> @date <br />
    @message
}
```

Both codes display the same output.

### Output:

Today's date is: 28-02-2024  
Hello All!

Start an if-else condition with @ symbol. Even for a single statement, enclose the if-else code in braces {}.

Code Snippet 4 displays how to write the if-else condition.

## Code Snippet 4

```
@if(DateTime.IsLeapYear  
    (DateTime.Now.Year) )  
{  
    @DateTime.Now.Year @: is a leap year.  
}  
else {  
    @DateTime.Now.Year @: is not a leap year.  
}
```

**Output:** 2024 is a leap year.

Code Snippet 5 displays a loop construct implemented in Razor.

## Code Snippet 5

```
@for (int i = 0; i < 4; i++) {  
    @i.ToString() <br />  
}
```

**Output:**

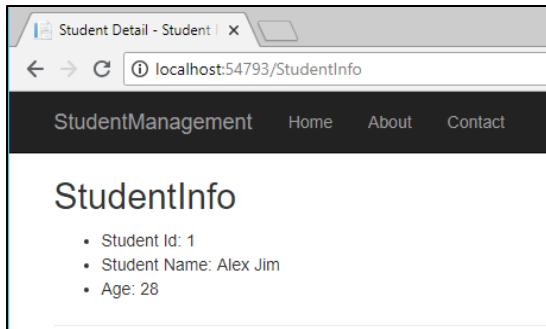
```
0  
1  
2  
3
```

Code Snippet 6 displays how to use a model object anywhere in the view by using @Model.

## Code Snippet 6

```
@model Student  
<h2>Student Detail:</h2>  
<ul>  
    <li>Student Id: @Model.StudentId</li>  
    <li>Student Name: @Model.StudentName</li>  
    <li>Age: @Model.Age</li>  
</ul>
```

Figure 7.4 displays the output, assuming that the Model object is populated with data.



**Figure 7.4: Output of Using model Object**

A variable can be declared inside a code block within brackets as shown in Code Snippet 7. These variables are then used in HTML with @ symbol.

### Code Snippet 7

```
@{
    string str = "";
    // assume
    if(i > 0) {
        str = "Hello World!";
    }
}
<p>@str</p>
```

### Output

Hello World!

Table 7.1 shows some of the most frequently used Razor commands.

Razor Command	Description
@if	Marks the beginning of the if condition.
@using	Allows using different packages, namespaces, and classes.
@try, catch, finally	Are used for exception handing.
@inherits	Gives full control over the inherited class.

**Table 7.1: Razor Commands**

There are several new enhancements in ASP.NET Core 7.0 and higher while implementing MVC Core Razor pages. These include MVC Model Binding improvements, support for C# 10, optimized Razor view compilation and rendering processes, and integration of additional security measures to mitigate common Web application vulnerabilities. They also include updates to development tools and Visual Studio 2022 extensions to enhance developer productivity, including improved IntelliSense support, enhanced debugging capabilities, and smoother integration with source control systems.

## 7.2 MVC Model Binding

MVC Model Binding maps HTTP request data with a model. Model Binding in ASP.NET MVC is a great illustration of how a small modification in the implementation of technology improves the calling of pages. Else, it would be tough to read an integer value from the query string of a URL. Model binder is like a bridge that maps the HTTP request with Controller action method.

Prior to .NET Core, there would be an exception while trying to attach a model. An exception is thrown when the controller method requires an integer and either no integer or no value that can be converted to an integer is given. If no integer is supplied, the Model Binding layer in ASP.NET Core automatically assigns the parameter number, the default value, such as 0 for an integer.

Microsoft added two new characteristics to ASP.NET Core that can be utilized with Model Binding. The goal is to determine whether binding occurs or not. The properties, such as `BindNever` and `BindRequired` are used. `BindNever` determines whether binding should never happen. `BindRequired` determines if binding is required once it has been applied. Therefore, if there is any request data that can be mapped to a number parameter, both the `Number` method argument and the `Number` property would receive the value in the default manner. A class property called `container` equipped with the `BindNever` attribute in ASP.NET Core, on the other hand, will not be susceptible to Model Binding because its value will be 0.

Figure 7.5 shows how Microsoft Visual Studio supports Model Binding.

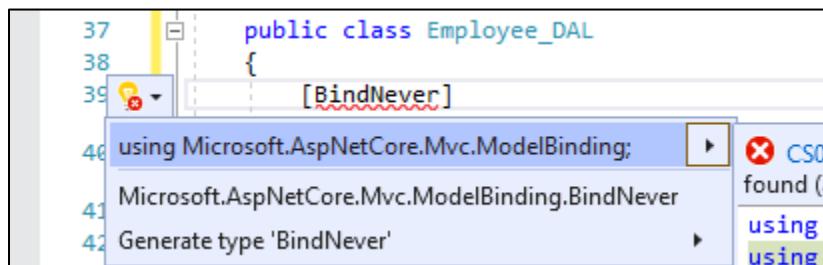


Figure 7.5: Model Binding

Code Snippet 8 displays the use of `BindNever`.

### Code Snippet 8

```
public class Employee.DAL
{
    [BindNever]
    public int Number { get; set; }
    public int Code { get; set; }
}
```

## 7.3 C# Record Type

C# record types can be used for data transfer between the caller (Controller) and sender (API). These are used with Model Binding in an MVC controller or a Razor page. This is for efficient communication.

Code Snippet 9 shows an example where a record `Movies` is created. `MoviesController` is the class in Controller file that uses the record `Movies` with Model Binding.

### Code Snippet 9

```
namespace WebApplications5.Controllers {
    public record Movies(string MoviesCode, string MoviesName,
    int[] MoviesBudget);
    public class MoviesController : Controller {
        public IActionResult Index() => View();
        [HttpPost]
        public IActionResult Index(Movies mv)
        {
            return View();
        }
    }
}
```

Code Snippet 10 shows form validation in cshtml file as a view.

### Code Snippet 10

```
Index.cshtml
@{
    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>
<table>
    <tr>
        <td>
            <input asp-for="MoviesCode" />
            <span asp-validation-for="MoviesCode" />
        </td>
        <td>
            Movie Name:
            <input asp-for="MoviesName" />
            <span asp-validation-for="MoviesName" />
        </td>
    </tr>
</table>
```

```

</td>
<td>
    Movie Budget:
    <input asp-for="MoviesBudget" />
    <span asp-validation-for="MoviesBudget" />
</td>
</tr>
</table>
@model Movie
Movie Code:

```

## 7.4 Text.Json

In ASP.NET Core, JavaScript Object Notation (JSON) supports UTF-8 text encoding and serialization library for converting .NET object types to a JSON string or vice versa. It is found under `System.Text.Json` namespace and has following objects:

<b>JsonSerializer</b>	This class allows serialization and deserialization of .NET objects to and from JSON.
<b>JsonDocument</b>	This class allows for better structural content management rather than data values being instantiated.
<b>JsonElement</b>	This class refers to a JSON value in a <code>JsonDocument</code> .
<b>Utf8JsonWriter</b>	This class provides API for writing UTF-8 encoded JSON content. Writes the text with no-cache.
<b>Utf8JsonReader</b>	This class provides API for reading UTF-8 encoded JSON content. Reads text without caching.

Code Snippet 11 creates a class `Employee` as part of an ASP.NET Core Web application.

### Code Snippet 11

```

public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

```

Code Snippet 12 serializes and deserializes the `Employee` object.

## Code Snippet 12

```
@{
    public IActionResult Index()
    {
        string Serialize;
        string DeSerialize;
        Employee obj = new Employee { FirstName = "Tyler",
                                      LastName = "King" };
        string json = JsonSerializer.Serialize(obj);
        Serialize = (json);
        var stringde = "{\"FirstName\":\"David\"}";
        Employee anotherObject =
            JsonSerializer.Deserialize<Employee>(stringde
        );
        DeSerialize = (anotherObject.FirstName);
        return View();
    }
}
```

In Code Snippet 12, Employee type object is initialized and then, serialized. Further stringde variable holds the string value. Then, again it is deserialized.

Figures 7.6, 7.7, and 7.8 display execution of Code Snippet 12 and adding breakpoints at Lines 36 and 34 respectively.



Figure 7.6: Serialization

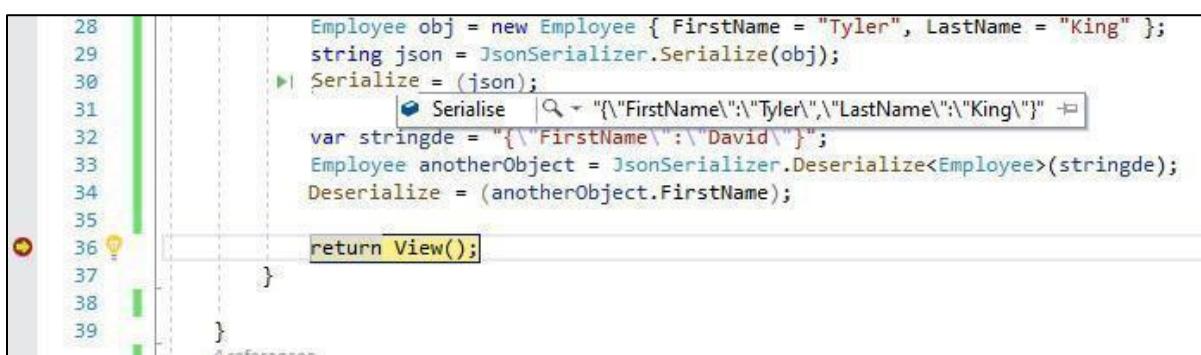
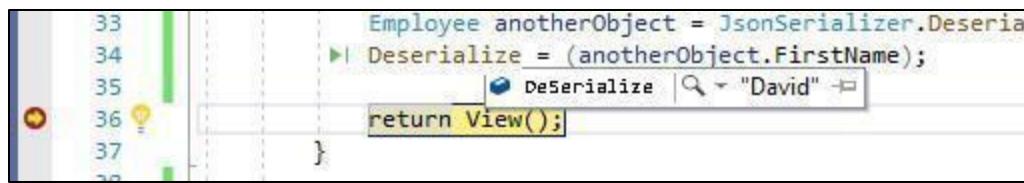


Figure 7.7: Serialize



**Figure 7.8: DeSerialize**

JsonSerializer has following features:

Serialization and Deserialization	Provides support for serializing and deserializing objects.
UTF 8 Data	Allows native processing of UTF-8 data.
JsonNamingPolicy	Specifies custom naming policies.
Asynchronous support	Built-in support for asynchronous serialization and deserialization.

## 7.5 Dump Debugging

Microsoft released a tool for obtaining heap dumps from a running .NET Core process as part of the diagnostics improvements in ASP.NET Core. One of the best ways to acquire and analyze operating system dumps, such as Windows and Linux dumps, is to use the dotnet-dump global tool.

Developers have to work a lot to improve the experience when working with dumps. The dotnet-dump tool is used to run SOS commands to investigate crashes and Garbage Collection (GC). However, this tool is not a debugger.

Commands are as follows:

**Dotnet-dump collect:** Collects the dump based on name, process id, and other parameters.

### Syntax

```
dotnet-dump collect [-h|--help] [-p|--process-id] [-n|--name] [--type]  
[-o|--output] [--diag]
```

**Dotnet-dump analyze:** Explores and analyzes the dump.

## Syntax

```
dotnet-dump analyze <dump_path> [-h|--help] [-c|--command]
```

## 7.6 Runtime Libraries

The foundation of all .NET class libraries comprises runtime libraries, which are extended through NuGet packages. Annotating runtime libraries for nullable reference types is crucial for improving performance and reducing the likelihood of encountering the `System.NullReferenceException` runtime exception.

Ensuring that nullable reference types are appropriately annotated and utilized for inputs and outputs is essential for compatibility with .NET Standard 2.1. To enable C# 10 features and add annotations for nullable reference types to a .NET Standard 2.0 library, include the following configuration in the project file:

```
<LangVersion>latest</LangVersion>
```

## 7.7 Garbage Collection

The Garbage Collector (GC) helps to automatically manage memory in the .NET framework. This means that code is not required for memory management. It manages memory allocation and the lifetime of objects. It reduces application downtime. This also helps reduce issues related to freeing objects or issues when an attempt is made to access objects that are freed. Manual release of objects is not required in GC. It also determines the time for garbage collection so that memory is released for objects that are not being used.

GC performs memory management through managed heaps. Heap refers to contiguous space in memory that is reserved at runtime when a process starts. GC looks into the roots of an application to determine the objects that are not required. In ASP.NET Core applications, when the application starts, GC allocates memory for initial heaps. It categorizes objects in generations from 0 to 2 based on the lifetime of an object. 0 is for objects that have the smallest lifetime. These are collected by GC frequently. Similarly, 1 is for objects that have a larger lifetime. These objects are not collected frequently.

# Summary

- ✓ Razor is a popular view-engine or a markup language used at the server-side.
- ✓ ASP.NET MVC supports the Razor view engine.
- ✓ Both HTML and server-side code using C# or Visual Basic (VB) are employed for writing Razor syntax.
- ✓ For an object to be immutable and behave like a value, it should be declared as a record.
- ✓ JSON supports UTF-8 text encoding and serialization library for converting .NET object types to a JSON string or vice versa.
- ✓ Any class libraries used by a project must be annotated for nullable reference types.
- ✓ Annotating runtime libraries for nullable reference types in .NET Standard 2.0 libraries is crucial for improving performance, reducing the likelihood of encountering null reference exceptions.
- ✓ The garbage collector is an automatic memory manager that maintains the lifetime of objects.

# Test Your Knowledge



1. Which of the following are the file extensions in the Razor view?

<b>A</b>	.vbhtml
<b>B</b>	.cshtml
<b>C</b>	html.cs
<b>D</b>	None of these

2. Which class allows structural content management?

<b>A</b>	JsonDocument
<b>B</b>	JsonElement
<b>C</b>	JsonSerializer
<b>D</b>	JsonReader

3. Which aspect is crucial for enhancing performance and reducing runtime exceptions in .NET Standard 2.0 libraries?

<b>A</b>	Annotating
<b>B</b>	Extending
<b>C</b>	Compiling
<b>D</b>	Bundling

4. Which of the following commands are used for exception handling in Razor code?

<b>A</b>	try
<b>B</b>	catch
<b>C</b>	Inherits
<b>D</b>	using

5. In an ASP.NET MVC application, a model class directly manages any input from the browser and will provide HTML output to the browser. (State True/False)

<b>A</b>	True
<b>B</b>	False

## Answers

1	A, B
2	A
3	A
4	A, B
5	B

## **Try It Yourself**

1. Create a new ASP.NET Core Web App in Visual Studio 2022, following the steps mentioned in the session. Name the project **MyRazorApp** and explore the default folder structure in the Solution Explorer.
2. Inside the **Views** folder of your project, create a new Razor page named **HomePage.cshtml**. Write HTML and Razor code to display a greeting message along with the current date and time.
3. In the **Views** folder, create a new folder named **Student**. Inside this folder, create a Razor page named **StudentInfo.cshtml**. Write code to display student information such as name, age, and course using Razor syntax.

# Session 8: .NET Core Architecture and Kestrel Web Server Implementation

## Session Overview

This session explains .NET Core architecture. It also describes some important architecture level pointers such as Main method in a Core application, the Kestrel server, the OWIN architecture, and the REST API architecture.

## Objectives

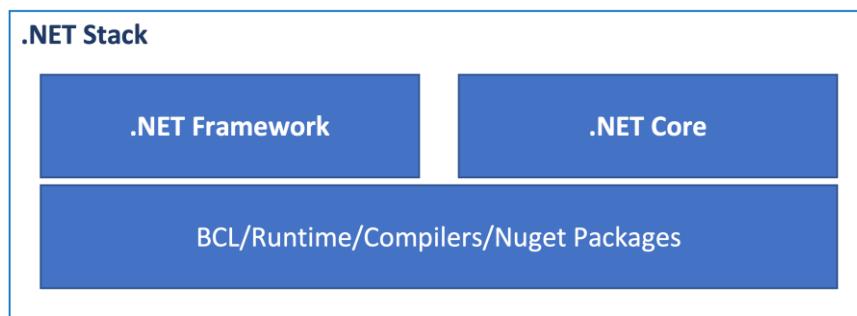
In this session, students will learn to:

- ✓ Describe .NET Core architecture
- ✓ Detail the execution of Main method
- ✓ Describe the Kestrel server
- ✓ Describe the OWIN architecture
- ✓ Explain REST API architecture
- ✓ Explain the difference between GraphQL API and REST API

## 8.1 .NET Core Architecture

.NET Core is an open-source development platform created and maintained by the .NET community. The objective of .NET Core is to bring together a diverse set of communities from other operating systems to make it a truly cross-platform development tool from Microsoft. Therefore, regardless of the operating system, .NET Core can be used as a development platform.

.NET Core in the .NET stack is represented in Figure 8.1.



**Figure 8.1: .NET Stack**

Until now, common runtime libraries, compilers, and NuGet Packages have been used by .NET Frameworks and libraries. Along with these shared packages, platform-specific libraries are also created. Definitions such as data types are stored in the Common libraries. As they rarely change, they serve as the foundation.

.NET Core is a platform where frameworks, such as ASP.NET Core and Universal Windows platform can use and extend the functionalities of the platform.

Features of .NET Core are as follows:

Cross-Platform	Microsoft extended its ideology build-once-run-anywhere to .NET Core so that it can build or run an application irrespective of the OS being used. It is open source and supports applications across all platforms, such as Windows, Linux, or Mac.
Command Line Interface (CLI)	.NET Core contains a variety of CLI tools, which helps in the local development. .NET core CLI is very lightweight. However, users also have the option of moving to an IDE.
Focus on Logic	.NET Core supports Continuous Integration and Continuous Deployment (CI/CD), so the focus remains on the logic.
Languages and IDEs	.NET Core applications and libraries can be coded in many languages through IDEs such as Visual Studio, VS Code, Sublime, Vim, and so on.
Compatibility and Support	Compatible with .NET Framework, .NET Standard, Mono APIs, and Xamarin. Both Microsoft and the community provide extensive support.
Device Focus	.NET Core allows development of apps across various domains, such as Mobile, Internet of Things (IOT), and gaming.
Representational State Transfer (RESTful) APIs	.NET Core does not support WCF. Therefore, there is a requirement for the creation of RESTful APIs.

Some architecture level aspects of importance are the `Main` method in a Core application, the Kestrel server, the Open Web Interface (OWIN) architecture, and the RESTful API.

## 8.2 Main Method in ASP.NET Core Applications

While creating an ASP.NET Core Web application, a file `Program.cs` is created by default in the application. This file contains a `Main` method. The `Main` method is defined as `public static void Main()`.

Figure 8.2 shows the `Main` method body.

```

12     public class Program
13     {
14         public static void Main(string[] args)
15         {
16             CreateHostBuilder(args).Build().Run();
17         }
18     }

```

**Figure 8.2: Main Method**

Similar to console applications, the `Main` method is named the entry point of the application in an ASP.NET Core application. Therefore, when an ASP.NET Core Web application is executed, the runtime looks for the `Main()` method and starts the execution. The `Main()` method then configures ASP.NET Core and starts it. Here, at this stage, the application becomes an ASP.NET Core Web application.

This method makes a call to the `CreateHostBuilder()` method. As shown in Figure 8.3, the `CreateHostBuilder()` method returns an object that implements the `IHostBuilder` interface.

```

public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
    ↗ IHostBuilder Program.CreateHostBuilder(string[] args)
1 reference

```

**Figure 8.3: CreateHostBuilder() Method**

Within the `Main()` method, on this `IHostBuilder` object, the `Build()` method is called which builds a Web host. Then, it hosts the ASP.NET Core Web application within that Web host. Finally, on the Web host, the `Run()` method is called to run the Web application and it starts processing the incoming HTTP requests.

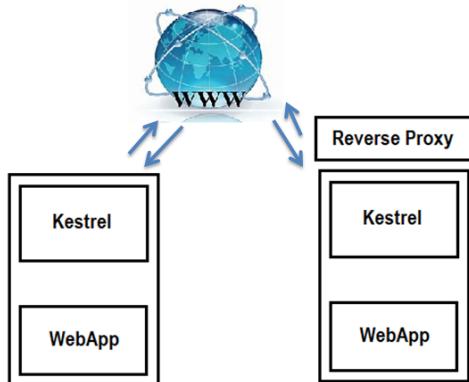
### 8.3 Kestrel Server

Kestrel is a cross-platform Web server for ASP.NET Core that works with all .NET Core platforms and versions. In ASP.NET Core, it serves as an internal server. It is included by default in the .NET Core application. This indicates that this server is compatible with all ASP.NET Core platforms and versions.

Dotnet.exe is the process that hosts the app in Kestrel. For the in-process hosting approach, Kestrel is not required. Kestrel can be used in following ways for out-of-process hosting:

- Kestrel can be used as a Web server that is accessible through the Internet.
- Kestrel can be used in conjunction with a reverse proxy server.

Figure 8.4 shows the to and from flow in the Kestrel server.



**Figure 8.4: Kestrel Server**

When the ASP.NET Core application is run using the .NET Core CLI, the Kestrel Web server is activated. It is responsible for handling and processing incoming HTTP requests.

Any ASP.NET Core Web application project, by default, has a method named `CreateHostBuilder`. This method calls `ConfigureWebHostDefaults` that uses the `UseKestrel` function internally. This is shown in Figure 8.5.

```
12  public class Program
13  {
14      public static void Main(string[] args)
15      {
16          CreateHostBuilder(args).Build().Run();
17      }
18
19      public static IHostBuilder CreateHostBuilder(string[] args) =>
20          Host.CreateDefaultBuilder(args)
21              .ConfigureWebHostDefaults(webBuilder =>
22              {
23                  webBuilder.UseStartup<Startup>();
24              });
25  }
```

**Figure 8.5: ConfigureWebHostDefaults Method**

Furthermore, the Kestrel server takes advantage of several configurations that are offered by environment variables, `appsettings.json` files, and even coding medium. When a Web program is running, it focuses on URLs that have been moved. Kestrel tracks two URLs by default if no configuration is supplied.

These URLs are as follows:

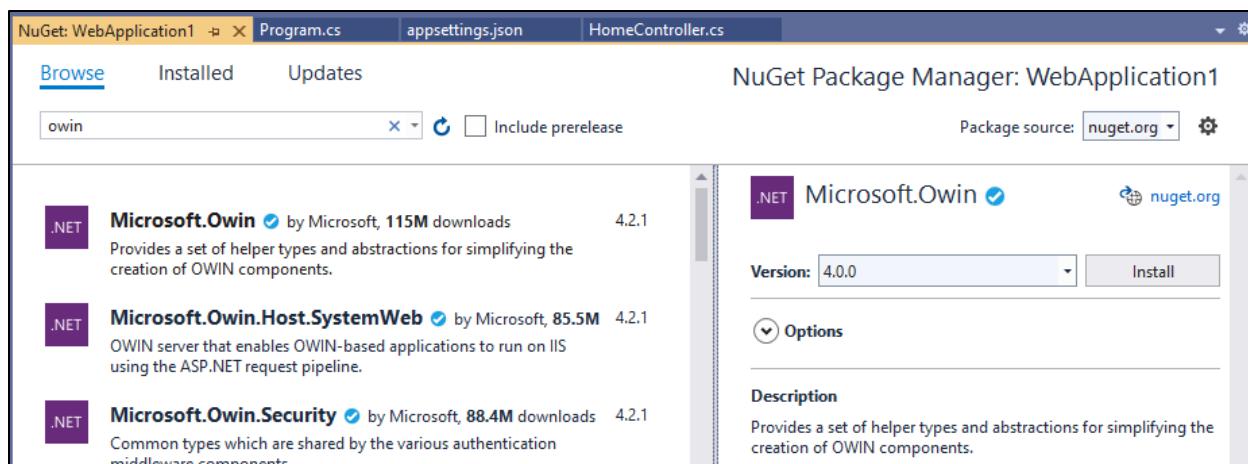
- <http://localhost:5000>
- <https://localhost:5001> (If the local development certificate is available)

## 8.4 OWIN Architecture

OWIN for .NET is an open-source framework that provides interface requirements to allow Web servers and applications to communicate.

Web programs can be isolated from Web servers using OWIN. It establishes a common method for middleware to handle requests and responses in a pipeline. OWIN-based applications, servers, and middleware may communicate with ASP.NET Core applications and middleware.

Support for OWIN can be found in NuGet Package Manager. Figure 8.6 shows the NuGet Package Manager.



**Figure 8.6: NuGet Package Manager**

ASP.NET is heavily reliant on IIS for deployment. This dependency on IIS restricts the mobility of ASP.NET applications. Hence, OWIN was created to make ASP.NET more modular by reducing dependencies and creating a loosely linked framework.

## 8.5 RESTful Web API

Representational State Transfer (REST) is an architectural approach for establishing standards among Web-based applications allowing them to communicate easily. Components are considered as resources and these resources can be accessed using a common interface that uses HTTP standard methods.

Although REST is not exactly related to HTTP, it is frequently associated with it. Built on the REST architecture, RESTful API is a lightweight, maintainable, and scalable service. It exposes API from an application to the calling client in a secure, standardized, and stateless manner.

Table 8.1 shows the four core HTTP verbs used in requests to communicate with resources in a REST system.

Command	Purpose
GET	To access a resource in a read-only mode
POST	To create a new resource
DELETE	To delete a resource
PUT	To update an existing resource or create a new one.

**Table 8.1: Core HTTP Attributes in REST System**

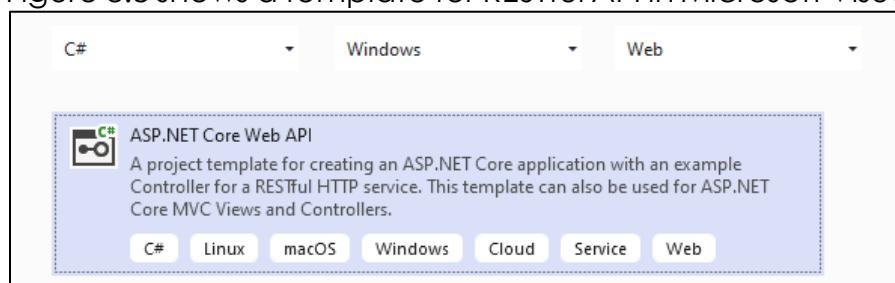
Figure 8.7 depicts how RESTful API interfaces with an application and resources.



**Figure 8.7: RESTful Web API**

RESTful API is the most effective way of accessing resources that are in different environments. For example, a client may require permission to access documents, videos, or images stored on a server. RESTful services will set up or define how these resources can be accessed.

Microsoft Visual Studio 2022 also supports the creation of a RESTful API based application. Figure 8.8 shows a template for RESTful API in Microsoft Visual Studio 2022.



**Figure 8.8: RESTful API Template**

In Code Snippet 1, the `[HttpGet]` attribute, which is a Get-based command, is used.

## Code Snippet 1

```
[HttpGet]
public IEnumerable<WeatherForecast> Get()
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new
        WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

When a `Get` method is used to fetch this service, after some basic operations, the resulting data appears in an array format. Figure 8.9 shows the result.

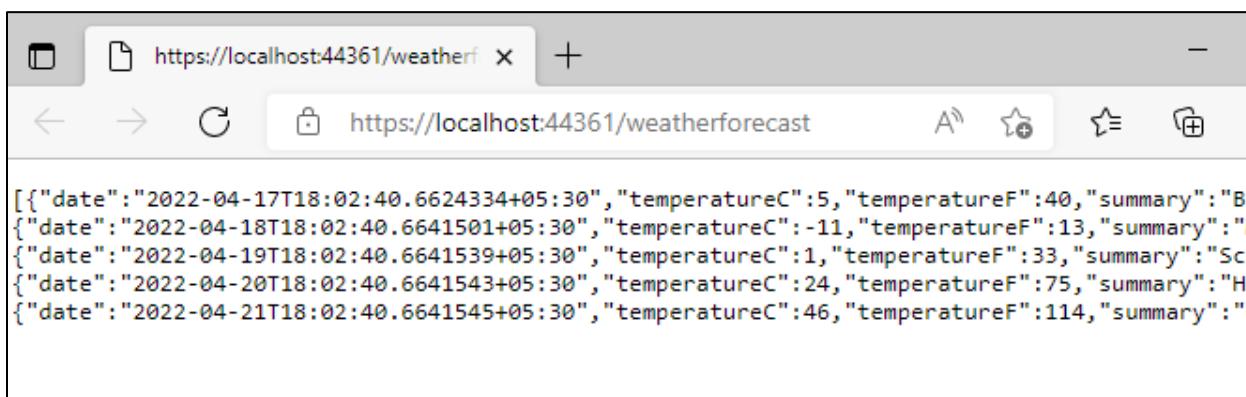


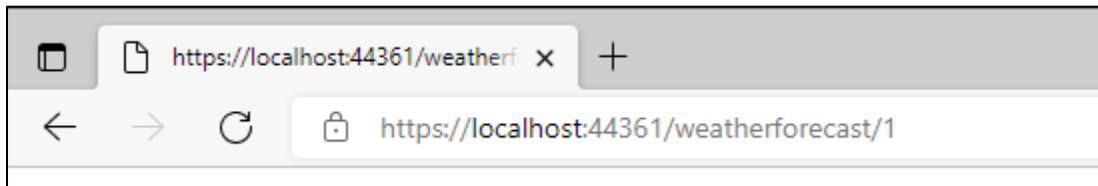
Figure 8.9: Output of Code Snippet 1

The `Get` method can also be modified to accept an argument as shown in Code Snippet 2.

## Code Snippet 2

```
[HttpGet("{id}")]
public IEnumerable<WeatherForecast> Get(int id)
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new
        WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

This modification helps to provide the argument from the browser as shown in Figure 8.10 while requesting the service.



**Figure 8.10: Argument from the Browser**

## 8.6 Difference Between GraphQL API and REST API

GraphQL is a powerful query language for APIs and a runtime environment for executing those queries with your existing data. Developed by Facebook in 2012 and later open-sourced in 2015, GraphQL has gained widespread adoption in the Web development community due to its flexibility, efficiency, and ability to address common challenges faced by traditional REST APIs.

### 8.6.1 Exploring GraphQL

Key concepts of GraphQL include:

- **Declarative Data Retrieval**

GraphQL empowers clients to request precisely the data they require. In contrast to REST, where servers dictate the response structure, clients articulate their data necessities in a query, obtaining only the requested information.

- **Unified Endpoint**

Unlike REST's typical approach of employing multiple endpoints for distinct resources, GraphQL generally exposes a solitary endpoint. This consolidation minimizes network requests and streamlines client-server communication.

- **Strongly Typed Schema**

GraphQL APIs are delineated by a schema that explicitly outlines the data types available for querying and their relationships. This schema serves as a contractual agreement between clients and servers, furnishing a lucid and organized interface.

- **Real-time Data Integration via Subscriptions**

GraphQL facilitates real-time data updates using subscriptions. Clients can subscribe to specific events and the server dispatches updates to subscribed clients when relevant data alterations occur.

- **Hierarchical Query Structure**

GraphQL queries adopt a hierarchical organization mirroring the structure of the response data. This hierarchical arrangement simplifies query comprehension, modification, and expansion for developers.

### 8.6.2 GraphQL Components

GraphQL components are listed in Table 8.2.

Component Name	Description
Queries	Clients use queries to request specific data from the server.
Mutations	Mutations are used to modify data on the server.
Subscriptions	Subscriptions enable real-time data updates by allowing clients to listen for specific events.
Types and Fields	GraphQL schemas define types and fields representing the data structure. Types can have fields and each field can return a specific type.
Resolvers	Resolvers determine how the server fulfills a query by specifying the logic to retrieve or manipulate data for each field.

**Table 8.2: Components of GraphQL**

### 8.6.3 Comparing Characteristics of GraphQL APIs and REST APIs

Table 8.3 shows differences between GraphQL and REST APIs.

Feature	GraphQL	REST API
Data Fetching	Allows clients to request specific data in a single query, reducing over-fetching and under-fetching issues	Typically involves multiple endpoints for different resources, leading to potential over-fetching or under-fetching.
Endpoint Structure	Typically exposes a single endpoint (for example, "/graphql")	Involves multiple endpoints, each dedicated to a specific resource or operation.
Response Format	Response structure mirrors the query shape and only includes requested data fields.	Server defines the response structure, which may include unnecessary or extraneous data.
Flexibility	Clients define the structure of the response based on their specific data requirements.	Server dictates the format of the response, potentially leading to over-fetching.
Versioning	No strict requirement for versioning as clients can request exactly what they require.	Versioning is often required to avoid breaking existing clients when API changes occur.

**Table 8.3: Difference Between GraphQL and REST APIs**

# Summary

- ✓ .NET Core is a platform where frameworks such as ASP.NET Core and Universal Windows Platform can use and extend the functionalities of the platform.
- ✓ Cross-Platform and architecture, Command Line Tools, Flexible CI/CD, Languages and IDEs, and Compatibility and Support are some of the features of .NET Core.
- ✓ The Main method is the entry point of the application in the ASP.NET Core application, which starts the execution from there.
- ✓ When the Main() method configures ASP.NET Core, the application becomes an ASP.NET Core Web application.
- ✓ Kestrel is a cross-platform Web server for ASP.NET Core that works with all .NET Core platforms and versions.
- ✓ OWIN for .NET is an open-source framework that provides interface requirements to allow Web servers and applications to communicate.
- ✓ OWIN-based applications, servers, and middleware may communicate with ASP.NET Core applications and middleware.
- ✓ REST is an architectural approach for establishing standards among Web-based applications, allowing them to communicate more easily.
- ✓ Built on the REST architecture, RESTful API is a lightweight, maintainable, and scalable service.
- ✓ GraphQL allows clients to request specific data in a single query, reducing over-fetching and under-fetching issues whereas REST typically involves multiple endpoints, leading to potential over-fetching or under-fetching of data.
- ✓ GraphQL response structure mirrors the query shape and includes only requested data fields whereas REST servers dictate the response structure, potentially including unnecessary or extraneous data.

# Test Your Knowledge



1. Which is the entry point of the application in the ASP.NET Core, where the execution starts?

<b>A</b>	CreateHostBuilder
<b>B</b>	ASP.NET
<b>C</b>	Main
<b>D</b>	Build()

2. Which is the internal Web server included by default in the .NET Core application?

<b>A</b>	Kestrel
<b>B</b>	OWIN
<b>C</b>	REST API
<b>D</b>	All of these

3. Which of these was created to make ASP.NET more modular by reducing dependencies and creating a loosely linked framework?

<b>A</b>	Kestrel
<b>B</b>	OWIN
<b>C</b>	RESTful API
<b>D</b>	All of these

4. \_\_\_\_\_ is an architectural approach for establishing standards among Web-based applications, allowing them to communicate more easily.

<b>A</b>	Kestrel
<b>B</b>	OWIN
<b>C</b>	REST API
<b>D</b>	RESTful API

5. Which of these is a lightweight, maintainable, and scalable service that exposes API from an application to the calling client in a secure, standardized, and stateless manner?

<b>A</b>	Kestrel
<b>B</b>	OWIN
<b>C</b>	REST API
<b>D</b>	RESTful API

## Answers

1	C
2	A
3	B
4	C
5	D

## **Try It Yourself**

1. Create a simple GraphQL query to fetch information about a hypothetical 'User' resource. Specify the fields you want to retrieve.
2. Define RESTful endpoints for a blog application that supports operations such as retrieving all posts, creating a new post, and updating an existing post.

# Session 9: Onion Architecture in ASP.NET Core – I

## Session Overview

This session explains inversion of control, dependency inversion principle, and Onion Architecture. It details different layers in the Onion Architecture. It also describes the advantages of Onion Architecture.

## Objectives

In this session, students will learn to:

- ✓ Explain inversion of control, dependency inversion principle, and Onion Architecture
- ✓ Describe the Onion Architecture layers
- ✓ Define Dependency Inversion Principle
- ✓ Explain the implementation of data access and repositories in the infrastructure layer
- ✓ Explain advantages of the Onion Architecture

## 9.1 Onion Architecture Concepts

Onion architecture is an architecture that was introduced in 2008. It was introduced to solve coupling issues in application design. Major challenges of traditional designs were tight coupling and concern for partition. Tight coupling and loose coupling are explained as follows:

### Tight Coupling:

In object programming, when a class has a strong dependency on other classes, then any kind of change in one object will affect the other classes. This is known as tight coupling. The impact may not be high small programs. However, it is challenging in large programs.

### Loose Coupling:

In loose coupling, two objects are independent of one another. One object can use another object without being dependent on it. The changes in one object do not affect the other. The design helps in eliminating the interdependencies between the system components. With loose coupling, there is no necessity to implement the changes of one component in others.

To address these challenges and issues that arise from three-tier and N-tier structures, the Onion Architecture was introduced. The Onion Architecture approach helps to develop a loosely coupled application.

It is important to understand the basic design principles on which Onion Architecture is based. These are Inversion of Control (IoC) and Dependency Inversion Principle (DIP).

### **9.1.1 Inversion of Control**

In object-oriented programming, many dependencies are generated between objects and components which becomes difficult to manage. Therefore, there are certain patterns and principles that are followed for designing the applications. Inversion of Control (IoC) is one such principle. It is a high-level design principle and provides some guidelines for object-oriented design.

As per IoC, the flow of controls is inverted in an object-oriented design. Here, controls mean the additional responsibilities of a class. This helps in creating loosely connected classes so that the objects are independent and there is no interdependency. When the classes are loosely coupled and the changes in one class do not impact the other, it is easy to test, maintain, and extend the program.

As per traditional approach, code makes calls to a library whereas as per IoC, a framework is used to define control of flow. IoC is implemented through Dependency Injection (DI). DI is a pattern whereas IoC is a principle. DI uses a framework for passing parameters and setting the properties. There is a built-in IoC container for implementing DI in ASP.NET Core.

Onion Architecture makes use of IoC.

### **9.1.2 Dependency Inversion Principle (DIP)**

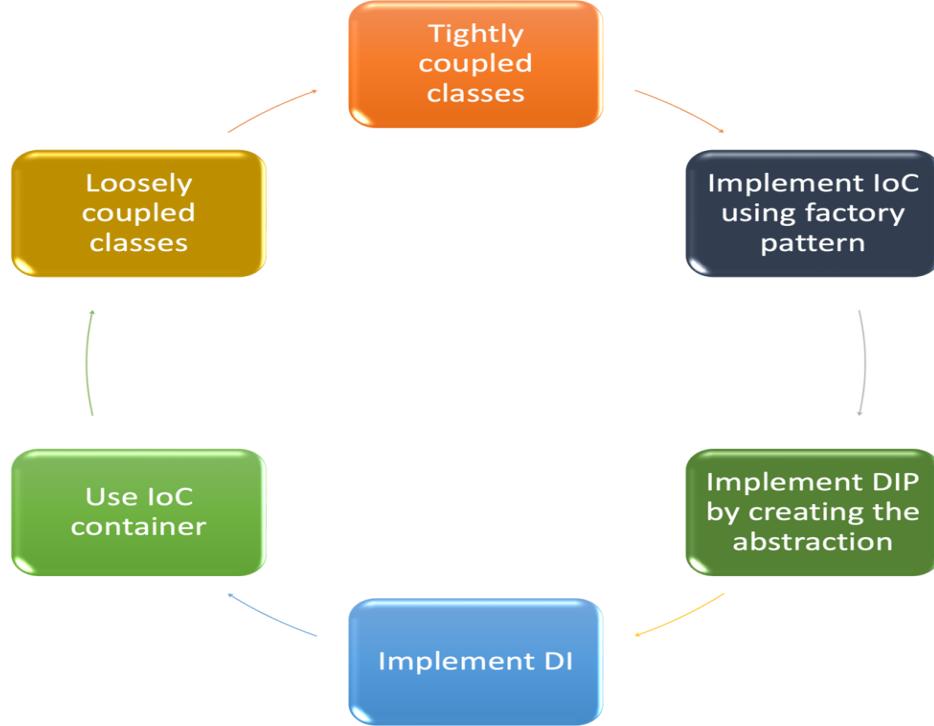
DIP stands as one of the fundamental principles within the Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion principles (SOLID), coined by Robert C. Martin to guide object-oriented design. This principle aims to foster development of maintainable and scalable software by advocating a flexible and loosely coupled architecture. DIP is used with IoC to achieve loose coupling. DIP removes the dependency between high-level modules or classes and low-level modules or classes. As per DIP, the classes should not be concrete so that they can be loosely coupled. Hence, abstraction, which means something that is not concrete, is used. Abstraction ensures both high-level and low-level classes depend on non-concrete interfaces or abstract classes. Abstractions should determine the details but not be specific.

To implement IoC and DIP for tightly coupled classes, following guidelines are followed:

- Implement IoC
- Implement DIP by creating an abstraction
- Implement dependency injection

- Use containership
- Create loosely coupled classes

Figure 9.1 represents the IoC and DIP implementation for tightly coupled classes.



**Figure 9.1: IoC and DIP Implementation**

## 9.2 Why Onion Architecture?

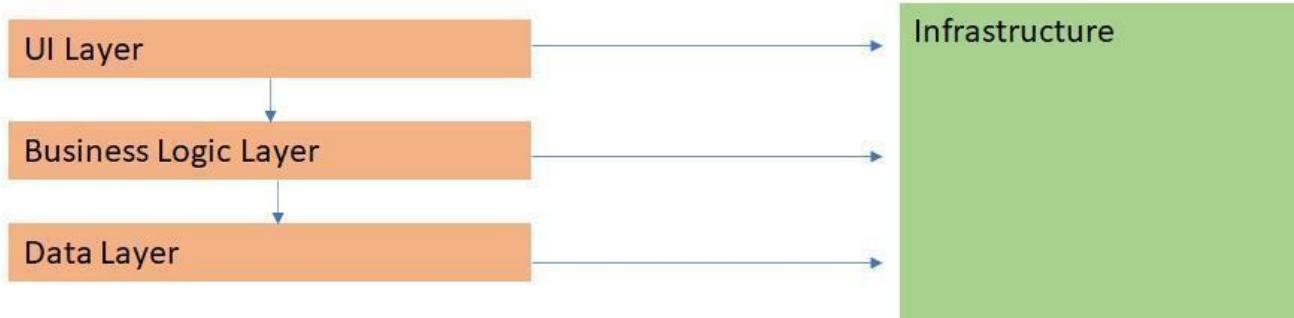
The aim of Onion Architecture was to build applications with less dependencies so that they can be managed and tested easily. Onion architecture solves the problems related to N-layer architecture. Though layering helps in separation of concerns, it also introduces issues of dependencies and coupling. Therefore, Onion Architecture was introduced by Jeffrey Palermo.

Onion architecture offers solution by providing several concentric layers that connect to the center. Domain layer is at the center. The Onion Architecture is based on the domain models rather than the data layers.

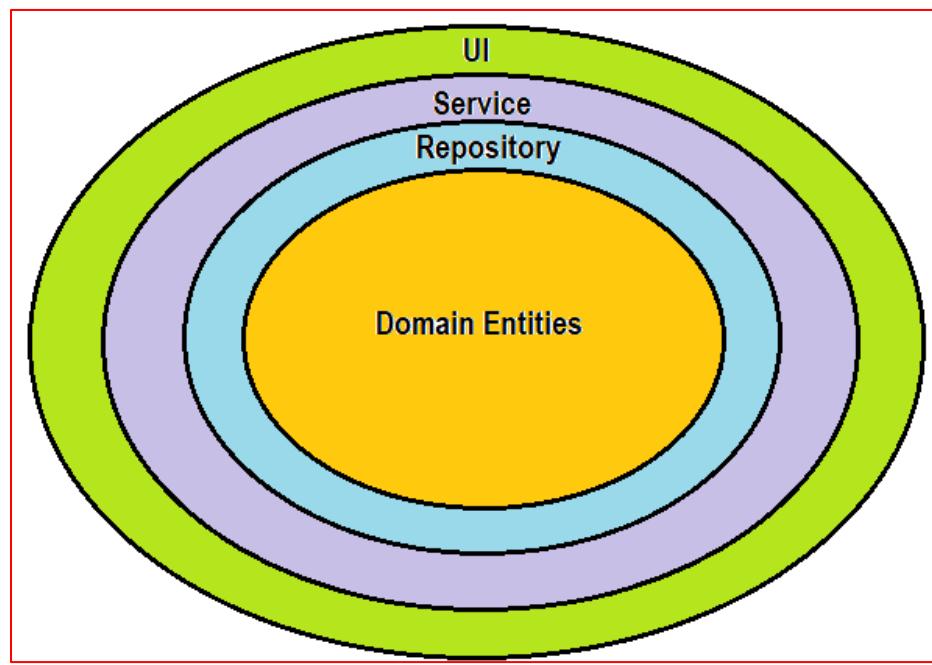
The concentric design layers of the Onion Architecture help to communicate with each other. This layered design helps to build better applications in terms of testability, practicality, and consistency. Also, the Onion Architecture provides solutions to common design problems.

### 9.3 Layers of the Onion Architecture

Figures 9.2 and 9.3 display traditional architecture and different layers of the Onion Architecture respectively.



**Figure 9.2: Traditional Architecture**



**Figure 9.3: Layers of Onion Architecture**

The Domain layer is at the center. The number of layers in the architecture may change depending on the application. However, the domain is always at the center. The outer layers are layers that may require changes often, for example, UI.

Following are different layers of the Onion Architecture.

- **Domain Entities Layer:** It is the deepest level of the Onion Architecture containing all application domain entities. Here, domain entities are the database models generated using a code-first approach.
- **Repository Layer:** The repository layer is between the services and the model objects. All database migrations and application data context object communication occur here. It consists of a read and write data access pattern for the database.
- **Service Layer:** This layer, which consists of exposable APIs, is responsible for the communication between the Repository layer and the main project. This layer also contains an entity's business logic and the interfaces are maintained stand-alone from their implementation for loose coupling and concern separation.
- **UI (Web/Unit Test) Layer:** The user interface is nothing more than a front-end program that communicates with the API.

## 9.4 Exploring DIP

DIP consists of two key principles:

- 1. High-level modules should not depend on low-level modules. Both should depend on abstractions.**
  - High-level modules are modules that contain the main business logic or application-specific functionality.
  - Low-level modules are modules that deal with the details or implementations.
- 2. Abstractions should not depend on details. Details should depend on abstractions.**
  - Abstractions refer to interfaces or abstract classes that define the contract or behavior without specifying the implementation details.
  - Details are the concrete implementations that fulfill the contract specified by the abstractions.

### 9.4.2 Practical Examples of Implementing DIP

Developers can implement the Dependency Inversion Principle (DIP) in practice by designing their software to depend on abstractions (interfaces or abstract classes) rather than concrete implementations.

Here are some practical examples of implementing DIP in a .NET Core application:

### 1. Service Abstraction:

Suppose developers have a high-level module that requires to perform logging, but they want to decouple it from specific logging implementations. They can define an interface for logging as shown in Code Snippet 1.

#### Code Snippet 1

```
public interface ILogger
{
    void Log(string message);
}
```

Now, the high-level module can depend on abstraction which is shown in Code Snippet 2.

#### Code Snippet 2

```
public class HighLevelModule
{
    private readonly ILogger logger;

    public HighLevelModule(ILogger logger)
    {
        this.logger = logger;
    }

    public void PerformOperation()
    {
        // High-level logic
        logger.Log("Operation performed successfully.");
    }
}
```

Different logging implementations can then adhere to the `ILogger` interface, such as a file logger, database logger, or a console logger. The high-level module remains independent of these specific implementations.

## 9.5 Data Access and Repositories in the Infrastructure Layer

In modern software architecture, the infrastructure layer serves as the backbone of an application, orchestrating vital mechanisms required for its operation. This layer encompasses various responsibilities, including data access, external communications, and infrastructure-related concerns.

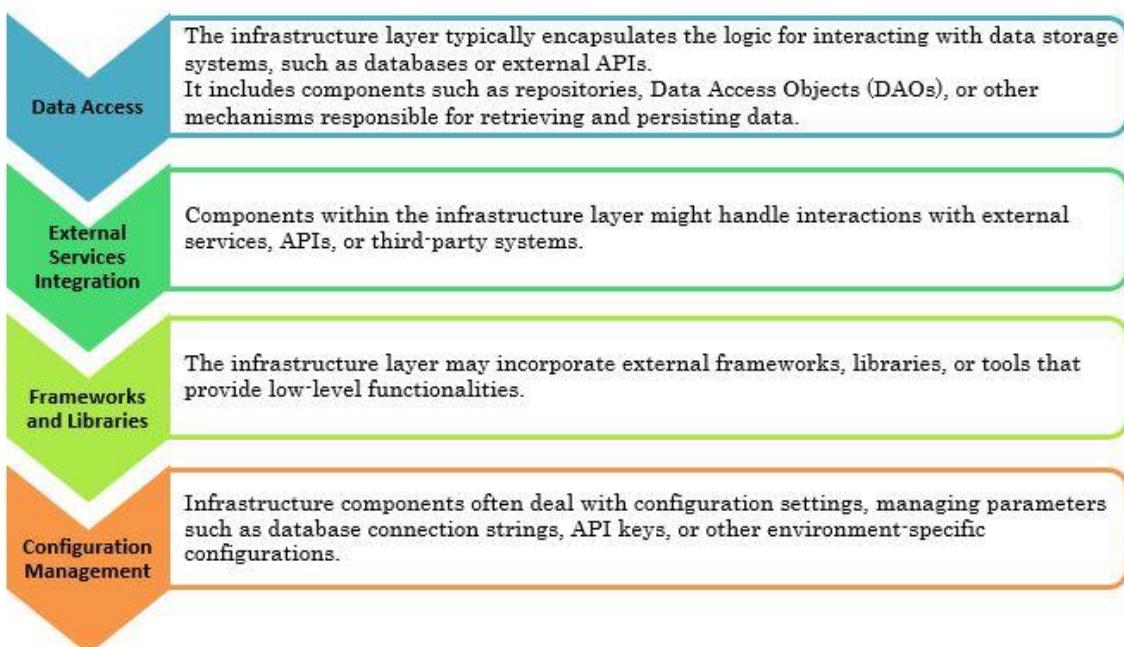
Let us delve deeper into it.

### 9.5.1 Infrastructure Layer

In the context of software architecture, the infrastructure layer represents the foundation of an application, dealing with the underlying mechanisms necessary for the application to function. It often includes components responsible for data access, external communications, and other infrastructure-related concerns. The infrastructure layer is one of the layers in the Onion Architecture, a design pattern that promotes a separation of concerns and a clear organization of code.

### 9.5.2 Responsibilities of the Infrastructure Layer

Responsibilities of the Infrastructure Layer are shown in Figure 9.4.



**Figure 9.4: Responsibilities of the Infrastructure Layer**

### 9.5.3 Implementation of Data Access Using Repositories

Implementing data access using repositories involves creating classes or components that abstract away the details of interacting with a data storage system and provide a clean and consistent interface for the rest of the application to work with. Following are the step-by-step guide on how to implement data access using repositories in a ASP.NET Core Web application.

#### 1. Define Repository Interface

Create an interface that defines the contract for data access operations. This interface should include methods for common Create, Read, Update, Delete (CRUD) operations as shown in Code Snippet 3.

### Code Snippet 3

```
public interface IRepository<T>
{
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
    T GetById(int id);
    IEnumerable<T> GetAll();
}
```

As a result of this code, the interface is defined.

## 2. Implement Repository

Create concrete implementations of the repository interface for each entity or data type. These implementations will contain the logic for interacting with the specific data storage system. For example, one can use Entity Framework for a SQL database with the code in Code Snippet 4.

### Code Snippet 4

```
public class SqlRepository<T> : IRepository<T> where T : class
{
    private readonly DbContext context;
    public SqlRepository(DbContext dbContext)
    {
        this.context = dbContext ?? throw new
ArgumentNullException(nameof(dbContext));
    }
    public void Add(T entity)
    {
        context.Set<T>().Add(entity);
        context.SaveChanges();
    }
    public void Update(T entity)
    {
        context.Set<T>().Update(entity);
        context.SaveChanges();
    }
    public void Delete(T entity)
    {
        context.Set<T>().Remove(entity);
        context.SaveChanges();
    }
    public T GetById(int id)
    {
        return context.Set<T>().Find(id);
    }
    public IEnumerable<T> GetAll()
```

```
{  
    return context.Set<T>().ToList();  
}  
}
```

Code Snippet 4 creates concrete implementations of the repository interface `IRepository`.

### 3. Inject Repository into Services

Use dependency injection to inject the repository into services or controllers. This allows the higher-level components of the application to use the repository without being tightly coupled to specific data access implementations as shown in Code Snippet 5.

#### Code Snippet 5

```
public class UserService  
{  
    private readonly IRepository<User> userRepository;  
  
    public UserService(IRepository<User> userRepository)  
    {  
        this.userRepository = userRepository ?? throw new  
ArgumentNullException(nameof(userRepository));  
    }  
  
    public void AddUser(User user)  
    {  
        userRepository.Add(user);  
    }  
  
    public IEnumerable<User> GetAllUsers()  
    {  
        return userRepository.GetAll();  
    }  
}
```

## 9.6 Advantages of the Onion Architecture

MVC design was able to solve the issues of concern partition, which was one of the major disadvantages of traditional architectures. In MVC, UI, business logic, and data access logic are all separated. However, tight coupling could not be resolved. Therefore, Onion Architecture was introduced. The Onion Architecture aims to retain an application's business logic, data access logic, and model in the middle while pushing dependencies outside. This results in coupling towards the center. Hence, it addresses both issues, the issue of separation of concerns and the tight coupling.

Following are the advantages of Onion Architecture:

- **Easy Maintenance:** Easier to maintain because all the codes are based on layers or the center.
- **Improved Testing:** Allows generation of unit tests for independent levels without affecting other components of an application.
- **Loose Coupling:** It creates a loosely linked application as the program's outer layer always communicates with the inner layer through interfaces.
- **Easy Implementation:** For anything that is to be taken from an external service, an interface is created that is taken care of by higher layers of the Onion Architecture. Internal layers do not rely on external layers.

Onion architecture works on following guidelines:

- The application follows an independent object model.
- Entire code can be compiled separately away from Infrastructure.
- Coupling is towards the center, Domain.
- Layers that are inner should be used to define the interfaces.
- Outer layers to be used to implement the interfaces.

In conclusion, Onion Architecture is easy to maintain as all the codes are separated or in the center. In this architecture, testing is at independent levels. It also ensures the outer layers connect to the inner layers through interfaces. Hence, they help in creating loosely coupled applications.

# Summary

- ✓ Inversion of Control (IOC) inverts the flow of controls in an object-oriented design.
- ✓ Dependency Inversion Principle (DIP) removes the dependency between the high-level modules or classes and low-level modules or classes.
- ✓ Onion Architecture is based on the principle of IOC and DIP and addresses two major challenges of traditional architecture – tight coupling and concern partition.
- ✓ It has different layers, namely, Domain entities layer, Repository layer, Service Layer, and UI Layer.
- ✓ It creates a loosely linked application as the program's outer layer always communicates with the inner layer through interfaces.
- ✓ The Onion Architecture retains an application's business logic, data access logic, and model in the middle while pushing dependencies far outside.
- ✓ Data access and repositories in the infrastructure layer involves creating modular and reusable components that abstract the complexities of interacting with a data storage system.

# Test Your Knowledge



1. Which problems of traditional architecture does the Onion Architecture address?

<b>a</b>	Tight Coupling
<b>b</b>	Loose Coupling
<b>c</b>	Concern Partition
<b>d</b>	None of these

<b>A</b>	a and b
<b>B</b>	a and c
<b>C</b>	b and c
<b>D</b>	a, b, and c

2. The Onion Architecture's layered design helps to build better applications in terms of \_\_\_\_\_.

<b>A</b>	Testability
<b>B</b>	Practicality
<b>C</b>	Consistency
<b>D</b>	Integrity

3. When a class is strongly tied to a concrete dependency, any kind of alterations in one object leads to modifications in several other objects, it is called \_\_\_\_\_.

<b>A</b>	Model-View-Controller
<b>B</b>	Tight Coupling
<b>C</b>	Concern Partition
<b>D</b>	Loose Coupling

4. Which layer is the core of the Onion Architecture?

<b>A</b>	Domain Entity Layer
<b>B</b>	Repository Layer
<b>C</b>	Service Layer
<b>D</b>	UI Layer

5. Which layer consists of exposable APIs in the Onion Architecture?

<b>A</b>	Domain Entity Layer
<b>B</b>	Repository Layer
<b>C</b>	Service Layer
<b>D</b>	UI Layer

## Answers

1	B
2	D
3	B
4	A
5	C

## **Try It Yourself**

1. Implement a Web application using Onion Architecture principles, focusing on the infrastructure layer's responsibilities such as data access and repository implementations.
2. Implement a simple repository pattern in C# using .NET Core. Create an interface `IRepository<T>` with methods for Create, Read, Update, and Delete (CRUD) operations on a generic type T. Then, create a concrete implementation of this interface for a specific entity (example., User). Write a sample usage code to demonstrate the CRUD operations on the User entity using the repository.
3. Implement dependency injection in an ASP.NET Core Web application. Define a service interface and its corresponding implementation. Configure dependency injection in the application's startup class and demonstrate the usage of the injected service in a controller or middleware.

# Session 10: Onion Architecture in ASP.NET Core - II

## Session Overview

This session describes the project structure of Onion Architecture and procedure to implement the four projects – Domain layer, Infrastructure layer, Service layer, and Onion Architecture Web API.

## Objectives

In this session, students will learn to:

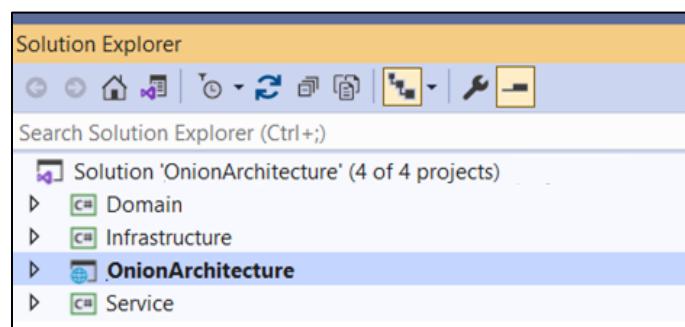
- ✓ Explain project structure for Onion Architecture
- ✓ Describe the process of implementation of the four projects – Domain layer, Infrastructure layer, Service layer, and Onion Architecture Web API
- ✓ List the security considerations
- ✓ Explain setting up monitoring and tracing for error detection and performance analysis

## 10.1 Project Structure of Onion Architecture

The concentric design of the layers of Onion Architecture helps to communicate easily. The layered design helps to build better applications. Applications are easy to test. This design also ensures consistency. Following are different layers of the Onion Architecture:

- Domain entities layer
- Repository layer
- Service layer
- UI layer

For each of these layers, a corresponding project is created in an ASP.NET Core application solution that implements Onion architecture. Figure 10.1 shows different layers as seen in Solution Explorer.

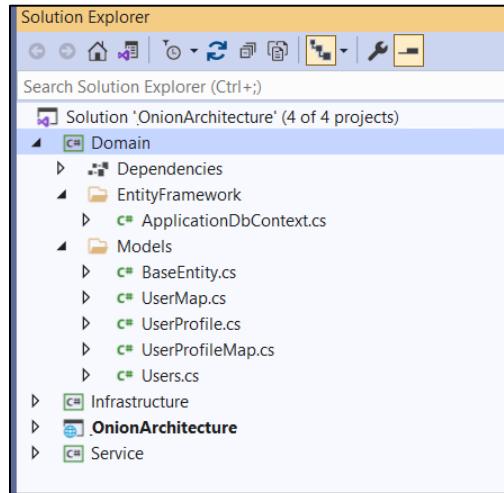


**Figure 10.1: Layers of Onion Architecture**

**Note:** For the purpose of this session, the example solution is also named OnionArchitecture, but one can give it any other meaningful and relevant name.

## 10.2 Domain Entities Layer

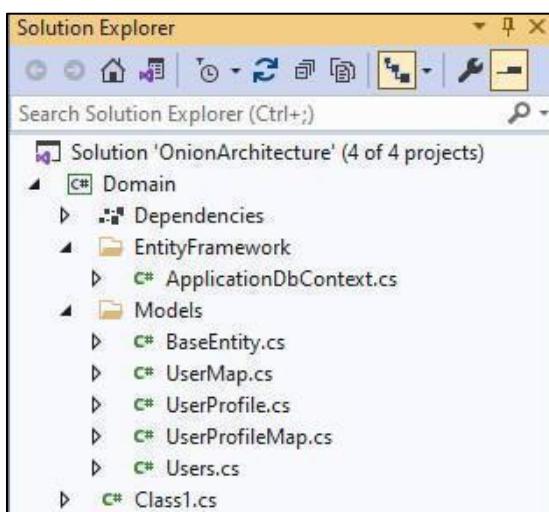
Domain Entities Layer is the most important layer of the Onion architecture, represented in Figure 10.2. It contains the class library, POCO, and configuration classes. It also helps in creating database tables.



**Figure 10.2: Domain Entity Layer**

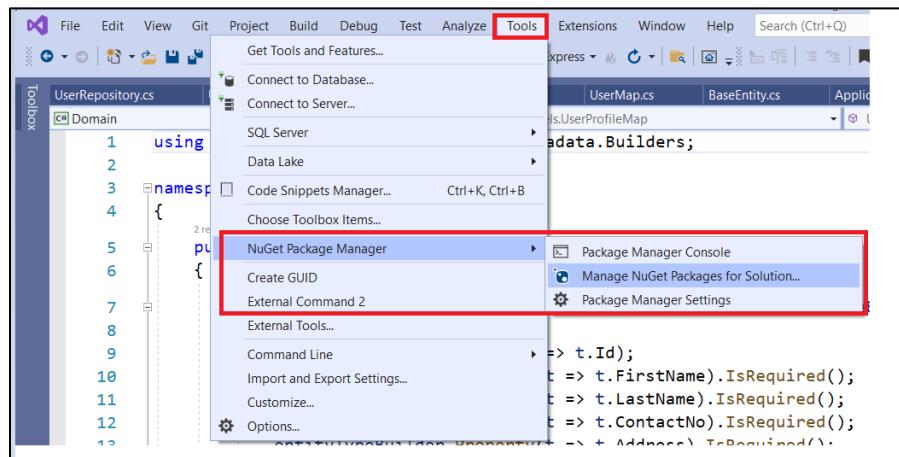
The process of implementation of Domain entity layer is as follows:

1. Create the Models and Entity Framework folders in the **Domain** folder. Create the class `ApplicationContext.cs` within the Entity Framework folder. Create five classes -  `BaseEntity`, `UserMap`, `UserProfile`, `UserProfileMap`, and `Users` within the Model folder as shown in Figure 10.3.



**Figure 10.3: Model and Entity Framework Folders**

2. Next, in the **Tools** menu, click **NuGet Package Manager → Manage NuGet Packages For Solutions** to install all the required packages in the domain layer as shown in Figure 10.4.



**Figure 10.4: NuGet Package Manager**

3. Select and install following projects individually as shown in Figure 10.5.
- Microsoft.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore.Design
  - Microsoft.EntityFrameworkCore.Tools
  - Microsoft.EntityFrameworkCore.SqlServer

 A screenshot of the NuGet Package Manager search results. The search bar at the top contains 'Microsoft.EntityFrameworkCore'. Below the search bar, there are four project entries, each with a '.NET' icon:
 

- Microsoft.EntityFrameworkCore** by Microsoft, 982M downloads. Description: Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, and database migrations. It can be used with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases.
- Microsoft.EntityFrameworkCore.Abstractions** by Microsoft, 947M downloads. Description: Provides abstractions and attributes that are used to configure Entity Framework Core.
- Microsoft.EntityFrameworkCore.Relational** by Microsoft, 942M downloads. Description: Shared Entity Framework Core components for relational database providers.
- Microsoft.EntityFrameworkCore.Analyzers** by Microsoft, 909M downloads. Description: CSharp Analyzers for Entity Framework Core.

**Figure 10.5: Entity Framework Projects**

4. In class `Users` in the Models folder, add code as shown in Code Snippet 1.

## Code Snippet 1

```
public class Users: BaseEntity {  
    public string UserName { get; set; }  
    public string Password { get; set; }  
    public string EMail{ get; set; }  
}
```

5. In class UserProfile in the Models folder, add code as shown in Code Snippet 2.

## Code Snippet 2

```
public class UserProfile : BaseEntity {  
    public string FirstName{ get; set; }  
    public string LastName{ get; set; }  
    public string Address { get; set; }  
    public string ContactNo{ get; set; }  
}
```

6. In class BaseEntity (which will be inherited by the Users class) in the Models folder, add code as shown in Code Snippet 3.

## Code Snippet 3

```
public class BaseEntity  
{  
    public Int64 Id { get; set; }  
    public DateTime ModifiedDate{ get; set; }  
    public string IPAddress { get; set; }  
}
```

7. Create a class called UserMap in the Models folder as shown in Code Snippet 4. UserMap class allows to define the primary key for the table.

## Code Snippet 4

```
public class UserMap {  
    public UserMap(EntityTypeBuilder<Users>  
entityTypeBuilder)  
    {  
        entityTypeBuilder.HasKey(t => t.Id);  
        entityTypeBuilder.Property(t =>  
            t.EMail).IsRequired();  
        entityTypeBuilder.Property(t =>  
            t.UserName).IsRequired();  
        entityTypeBuilder.Property(t =>
```

```

        t.Password).IsRequired();
    }
}

```

8. In class `UserProfileMap` in the Models folder, add code shown in Code Snippet 5. In Code Snippet 5, the ID is set as the primary key and the address field's length is specified as 100.

### Code Snippet 5

```

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

public class UserProfileMap {
    public UserProfileMap(EntityTypeBuilder<UserProfile>
entityTypeBuilder)
    {
        entityTypeBuilder.HasKey(t => t.Id);
        entityTypeBuilder.Property(t => t.FirstName).IsRequired();
        entityTypeBuilder.Property(t => t.LastName).IsRequired();
        entityTypeBuilder.Property(t => t.ContactNo).IsRequired();
        entityTypeBuilder.Property(t =>
t.Address).IsRequired().HasMaxLength(100);
    }
}

```

9. Create a folder called EntityFramework in the Domain layer and configure the context, that is, `ApplicationDbContext` as shown in Code Snippet 6.

### Code Snippet 6

```

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions opt) :
base(opt)
    { }
    protected override void OnModelCreating(ModelBuilder
oModelBuilder)
    {
        new UserMap(oModelBuilder.Entity<Users>());
        new
UserProfileMap(oModelBuilder.Entity<UserProfile>());
    }
}

```

10. Define the relationship for `Users` and `UserProfile` in the Domain layer and create the user class in the Models folder as shown in Code Snippet 7.

## Code Snippet 7

```
public class Users: BaseEntity {  
    public string UserName { get; set; }  
    public string Password { get; set; }  
    public string EMail{ get; set; }  
    public UserProfile userProfile { get; set; }  
}
```

11. Define the User class as a foreign key in the UserProfile class in the Models folder as shown in Code Snippet 8.

## Code Snippet 8

```
public class UserProfile : BaseEntity {  
    public string FirstName{ get; set; }  
    public string LastName{ get; set; }  
    public string Address { get; set; }  
    public string ContactNo{ get; set; }  
    public Users user{ get; set; }  
}
```

12. Modify the UserMap class in the Models folder and define UserProfile as a foreign key as shown in Code Snippet 9.

## Code Snippet 9

```
public class UserMap {  
    public UserMap(EntityTypeBuilder<Users>  
entityTypeBuilder)  
    {  
        entityTypeBuilder.HasKey(t => t.Id);  
        entityTypeBuilder.Property(t =>  
            t.EMail).IsRequired();  
        entityTypeBuilder.Property(t =>  
            t.UserName).IsRequired();  
        entityTypeBuilder.Property(t =>  
            t.Password).IsRequired();  
  
        entityTypeBuilder.HasOne(t =>  
            t.userProfile).WithOne(u =>  
            u.user).HasForeignKey<UserProfile>(x =>  
            x.Id);  
    }  
}
```

13. To build the project, open the main project – OnionArchitecture and add the reference in OnionArchitecture.csproj as shown in Code Snippet 10.

## Code Snippet 10

```
<ItemGroup>
    <ProjectReference Include="..\Domain\Domain.csproj" />
</ItemGroup>
```

14. Add the code in Code Snippet 11 in the appSettings.json.

## Code Snippet 11

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "ConnectionStrings": {
        "OnionConnection": "Data Source=DESKTOP-GMKPTTF\\SQLEXPRESS;Initial Catalog=OnionDB;Integrated Security=SSPI"
    },
    "AllowedHosts": "*"
}
```

15. In the Program.cs of OnionArchitecture project, add code given in Code Snippet 12. It is important to add the context from the configuration point of view. Replace ApplicationDbContext with actual DbContext class name.

## Code Snippet 12

```
using Domain.EntityFramework;
using Microsoft.EntityFrameworkCore;
. . .
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<ApplicationContext>(
    options => options.UseSqlServer(
        Configuration.GetConnectionString("OnionConnection"),
    b => b.MigrationsAssembly("OnionArchitecture")));
}
```

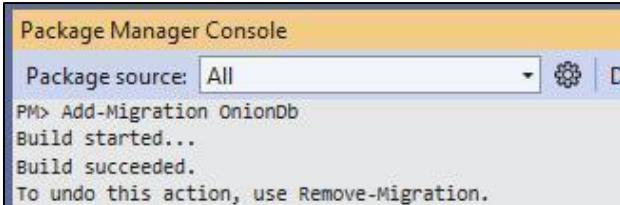
16. In the Package Manager Console, type the command Add-Migration OnionDb to start the database migration. An error as shown in Figure 10.6 appears.



```
Package Manager Console
Package source: All Default project: OnionArchitecture
PM> Add-Migration OnionDb
Build started...
Build succeeded.
Your startup project 'OnionArchitecture' doesn't reference Microsoft.EntityFrameworkCore.Design. This package is required for the Entity Framework Core Tools to work. Ensure your startup project is correct, install the package, and try again.
PM>
100 %
```

**Figure 10.6: Package Manager Console**

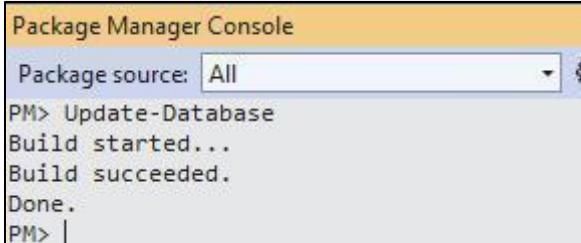
17. Add Microsoft.EntityFrameworkCore.Design package as per the error message.
18. Build again. The build is complete and a message appears as shown in Figure 10.7.



```
Package Manager Console
Package source: All
PM> Add-Migration OnionDb
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

**Figure 10.7: Build Successful Message**

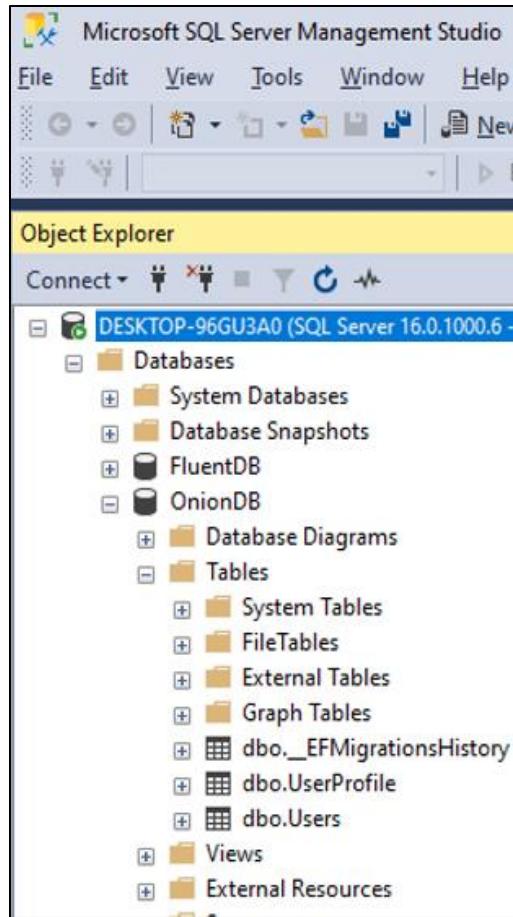
19. Run the command Update-Database on the same Package Manager Console as shown in Figure 10.8.



```
Package Manager Console
Package source: All
PM> Update-Database
Build started...
Build succeeded.
Done.
PM> |
```

**Figure 10.8: Update Database**

20. Open SQL Server Management Studio. Database and tables appear in it as shown in Figure 10.9.



**Figure 10.9: Database and Tables**

21. Open a new query window and insert one record into each table as shown:

```
INSERT INTO UserProfile (id, FirstName, LastName, Address,
ContactNo, ModifiedDate, IPAddress) VALUES (1, 'Tyler', 'king',
'LA', '0129-8276-353', '01/01/2022', '192.168.1.1')
```

```
INSERT INTO Users (UserName, Password, Email, ModifiedDate,
IPAddress) VALUES ('Tyler', 'King', 'tylerking@gmail.com',
'01/01/2022', '192.168.1.1')
```

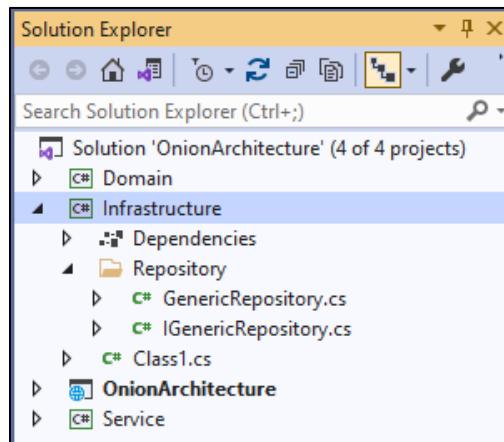
22. Ensure `Users` and `UserProfile` are linked to each other through the same ID.

### 10.3 Infrastructure Layer

The repository layer is the second-class library project. The repository layer implements the interface for the generic repository class. It also contains the `DbContext` class. Entity Framework source code creates a data access context class that inherits from the `DbContext` class.

Steps to work with this layer are as follows:

1. Create a project named Infrastructure in the same solution.
2. Create the Repository folder under Infrastructure and then, create an interface `IGenericRepository` and its corresponding class `GenericRepository` under the Repository folder as shown in Figure 10.10.



**Figure 10.10: Infrastructure Layer**

3. Add the reference of the Domain project in the infrastructure layer project as shown in Code Snippet 13.

### Code Snippet 13

```
<ItemGroup>
    <ProjectReference Include="..\Domain\Domain.csproj" />
</ItemGroup>
```

4. Create the `IGenericRepository` interface in the Repository folder as shown in Code Snippet 14.

### Code Snippet 14

```
public interface IGenericRepository<T> where T: BaseEntity
{
    IEnumerable<T> GetAll();
    T GetT(long id);
    void Insert(T entity);
    void Update(T entity);
    void Delete(T entity);
    void Remove(T entity);
    void SaveChanges();
}
```

5. In the GenericRepository class, complete the dependency injection and pass the ApplicationContext in the GenericRepository constructor as shown in Code Snippet 15.

### Code Snippet 15

```
using Domain.EntityFramework;
using Domain.Model;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;

public class GenericRepository<T> : IGenericRepository<T> where T
    : BaseEntity
{
    public readonly ApplicationContext dbContext;
    private DbSet<T> entities;

    public GenericRepository(ApplicationContext _dbContext)
    {
        dbContext = _dbContext;
        this.entities = dbContext.Set<T>();
    }
}
```

6. Define the DbSet with SetMethod as shown in Code Snippets 16 to 19.

### Code Snippet 16

```
public void Delete(T entity)    {
    if (entity == null)
    {
        throw new ArgumentNullException("Entity Missing");
    }
    else
    {
        entities.Remove(entity);
        dbContext.SaveChanges();
    }
}
```

### Code Snippet 17

```
public IEnumerable<T> GetAll()          {
    return entities.AsEnumerable();
}
public T GetT(long id)
```

```
{  
    return entities.SingleOrDefault(s => s.Id == id);  
}  
public void Insert(T entity)  
{  
    if (entity == null)  
    {  
        throw new ArgumentNullException("Entity Missing");  
    }  
    else  
    {  
        entities.Add(entity);  
        dbContext.SaveChanges();  
    }  
}
```

### Code Snippet 18

```
public void Remove(T entity) {  
    if (entity == null) {  
        throw new ArgumentNullException("Entity Missing");  
    }  
    else  
    {  
        entities.Remove(entity);  
    }  
}
```

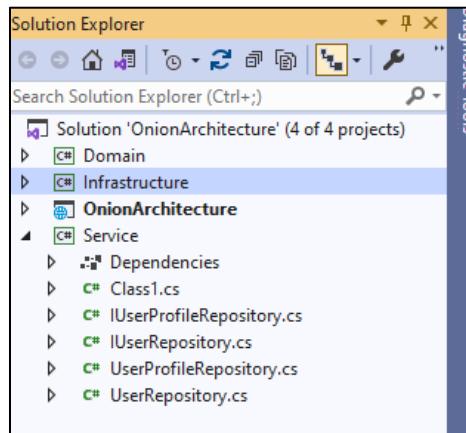
### Code Snippet 19

```
public void SaveChanges() {  
    dbContext.SaveChanges();  
}  
public void Update(T entity) {  
    if (entity == null) {  
        throw new ArgumentNullException("Entity Missing");  
    }  
    else  
    {  
        entities.Update(entity);  
        dbContext.SaveChanges();  
    }  
}
```

7. Finally, build the project.

## 10.4 Service Layer

The service layer contains the business logic and user interfaces as shown in Figure 10.11. The user interface and the data access logic communicate with each other. Consequently, it helps in creating loosely linked applications.



**Figure 10.11: Service Layer**

Steps to implement service layer are as follows:

1. Add the reference of the domain layer and infrastructure layer in the service layer as shown in Code Snippet 20.

### Code Snippet 20

```
<ItemGroup>
    <ProjectReference Include="..\Domain\Domain.csproj" />
    <ProjectReference
Include="..\Infrastructure\Infrastructure.csproj" />
</ItemGroup>
```

2. Create the interface **IUserRepository** in the service layer as shown in Code Snippet 21.

### Code Snippet 21

```
using Domain.Model;
using System.Collections.Generic;
public interface IUserRepository
{
    IEnumerable<Users> GetUsers();
    Users GetUser(long id);
    void InsertUser(Users user);
    void UpdateUser(Users user);
    void DeleteUser(long id);
}
```

3. Create the interface **IUserProfileRepository** in the service layer as shown in Code Snippet 22.

### Code Snippet 22

```
using Domain.Model;
public interface IUserProfileRepository
{
    UserProfile GetUserProfile(long id);
}
```

4. Create the class **UserProfileRepository** in the service layer as shown in Code Snippet 23.

### Code Snippet 23

```
using Domain.Model;
public class UserProfileRepository: IUserProfileRepository
{
    IGenericRepository<UserProfile> userProfileRepository;
    public UserProfileRepository(IGenericRepository<UserProfile> userProfileRepository)
    {
        userProfileRepository = userProfileRepository;
    }
    public UserProfile GetUserProfile(long id)
    {
        return userProfileRepository.GetT(id);
    }
}
```

5. Create the class **UserRepository** in the service layer with code as shown in Code Snippets 24 and 25.

### Code Snippet 24

```
public class UserRepository:IUserRepository
{
    IGenericRepository<Users> iuserRepository;
    IGenericRepository<UserProfile> iUserProfileRepository;

    public UserRepository(IGenericRepository<Users>
_iuserRepository, IGenericRepository<UserProfile>
_iUserProfileRepository)
    {
        iuserRepository= _iuserRepository;
        iUserProfileRepository = _iUserProfileRepository;
    }
}
```

```

    }
    public void DeleteUser(long id)
    {
        UserProfile userProfile =
iuserRepository.GetT(id);
        iuserRepository.Remove(userProfile);

        Users user = GetUser(id);
        iuserRepository.Remove(user);
        iuserRepository.SaveChanges();
    }
}

```

### Code Snippet 25

```

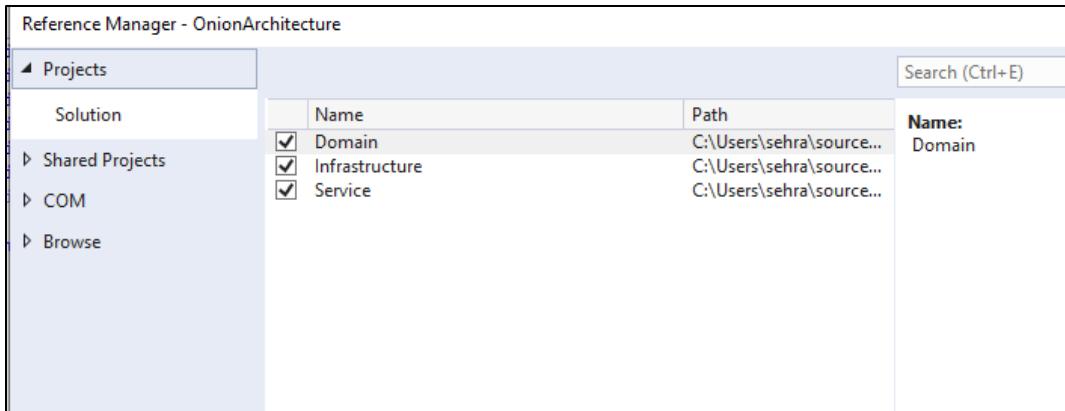
public Users GetUser(long id)
{
    return iuserRepository.GetT(id);
}
public IEnumerable<Users> GetUsers()
{
    return iuserRepository.GetAll();
}
public void InsertUser(Users user)
{
    iuserRepository.Insert(user);
}
public void UpdateUser(Users user)
{
    iuserRepository.Update(user);
}

```

## 10.5 Onion Architecture Web API

Onion Architecture Web API is the user interface layer. It is the entry point of the program and the exterior layer of the Onion Architecture. Users interact with this layer through a browser or Postman. It creates loosely linked applications with built-in dependency injection.

1. Add the references for the domain layer, infrastructure layer, and service layer in **OnionArchitecture** project as shown in Figure 10.12.



**Figure 10.12: Adding Reference - OnionArchitecture**

2. Create the **DTO** folder and add **UserDTO.cs** under the **DTO** folder as shown in Code Snippet 26.

### Code Snippet 26

```
public class UserDto
{
    public Int64 Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public string EMail { get; set; }
    public DateTime AddedDate { get; set; }
    public string ContactNo { get; set; }
}
```

3. Add the bolded code as shown in Code Snippet 27 in **Program.cs**.

### Code Snippet 27

```
...
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<EmployeeContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString(
    "OnionConnection"),
    b => b.MigrationsAssembly("OnionArchitecture")));
    services.AddScoped(typeof(IGenericRepository<>),
typeof(GenericRepository<>));
    services.AddTransient<IUserProfileRepository,
UserProfileRepository>();
    services.AddTransient<IUserRepository,
```

```
    UserRepository>();
    services.AddControllers();
}
```

4. Create the Controller class, `UserController`, as shown in Code Snippet 28.

### Code Snippet 28

```
using Domain.Model;
using Microsoft.AspNetCore.Mvc;
using Service;
using System.Collections.Generic;
using System.Linq;
public class UserController: ControllerBase
{
    IUserRepository iuserRepository;
    IUserProfileRepository iuserRepository;
    public UserController(IUserRepository
_iuserRepository,
    IUserProfileRepository _iuserRepository)
    {
        iuserRepository = _iuserRepository;
        iuserRepository = _iuserRepository;
    }
}
```

5. Add the `ListUsers()` method within the class as shown in Code Snippet 29.

### Code Snippet 29

```
[HttpGet]
    public ActionResult ListUsers()
    {
        List<Users> lstUser = new List<Users>();
        iuserRepository.GetUsers().ToList().ForEach(u =>
{
        Users user = null;
        UserProfile userProfile =
            iuserRepository.GetUserProfile(u.Id);
        user = new Users() -
        {
            Id = u.Id,
            UserName = u.UserName,
            EMail = u.EMail,
            Password = u.Password,
            IPAddress = u.IPAddress,
            ModifiedDate = u.ModifiedDate,
            userProfile = new UserProfile()
        }
    }
}
```

```

        {
            FirstName = userProfile.FirstName,
            LastName = userProfile.LastName,
            ContactNo = userProfile.ContactNo,
            Address = userProfile.Address,
            ModifiedDate = userProfile.ModifiedDate,
            IPAddress = userProfile.IPAddress,
            user = user,
            Id = u.Id
        }
    };
    lstUser.Add(user);
}
return Ok(lstUser);
}

```

6. Add the **CreateUser()** method as shown in Code Snippet 30.

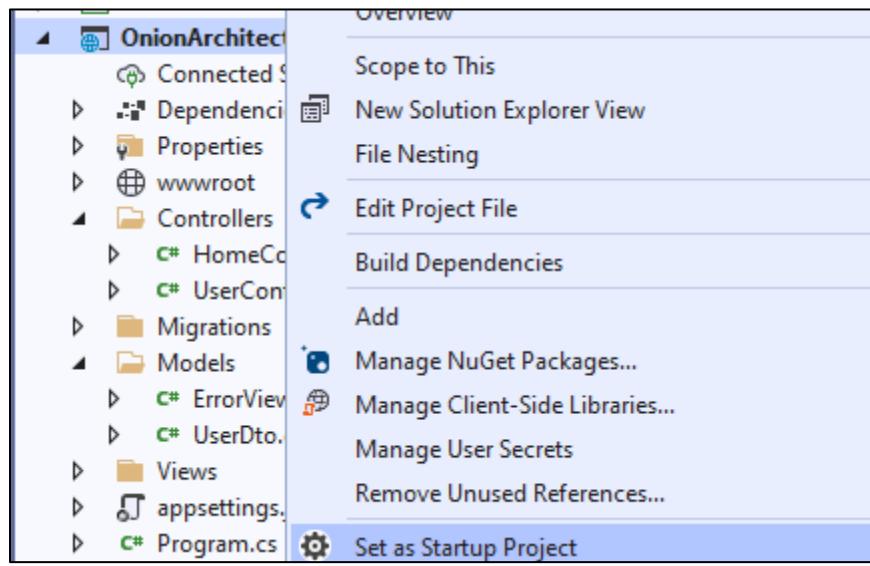
### Code Snippet 30

```

[HttpGet]
public int CreateUser(UserDto model)
{
    Users userEntity = new Users
    {
        UserName = model.UserName,
        EMail = model.EMail,
        Password = model.Password,
        ModifiedDate = DateTime.UtcNow,
        IPAddress =
Request.HttpContext.Connection.RemoteIpAddress
.ToString(),
        userProfile = new UserProfile
        {
            FirstName = model.FirstName,
            Address = model.Address,
            ContactNo = model.ContactNo,
            LastName = model.LastName,
            ModifiedDate = DateTime.UtcNow,
            IPAddress =
Request.HttpContext.Connection.RemoteIpAddress
.ToString(),
        }
    };
    iuserRepository.InsertUser(userEntity);
    return 1;
}

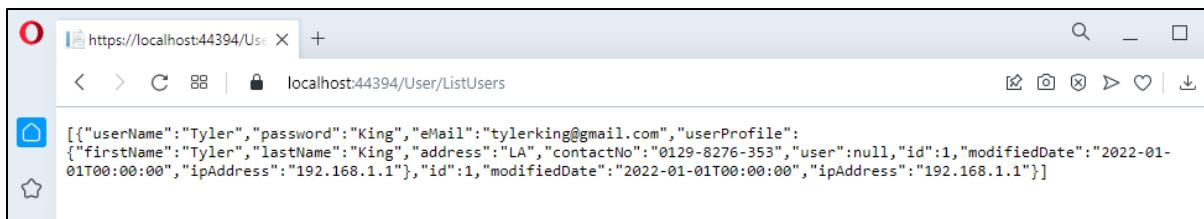
```

7. Right-click **OnionArchitecture** and select **Set as Startup Project** as shown in Figure 10.13.



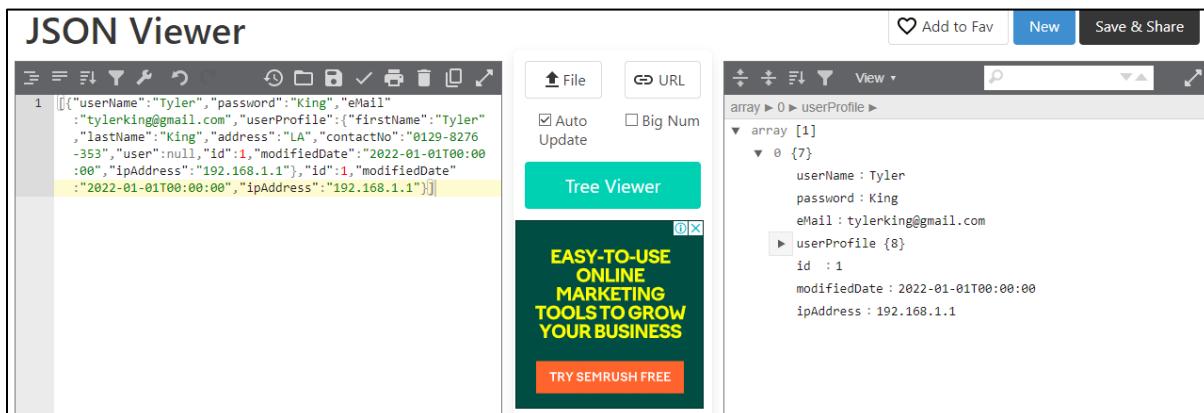
**Figure 10.13: Set as Startup Project**

8. Add appropriate views and run the application. The user details would be displayed as JSON data in the browser as shown in Figure 10.14.

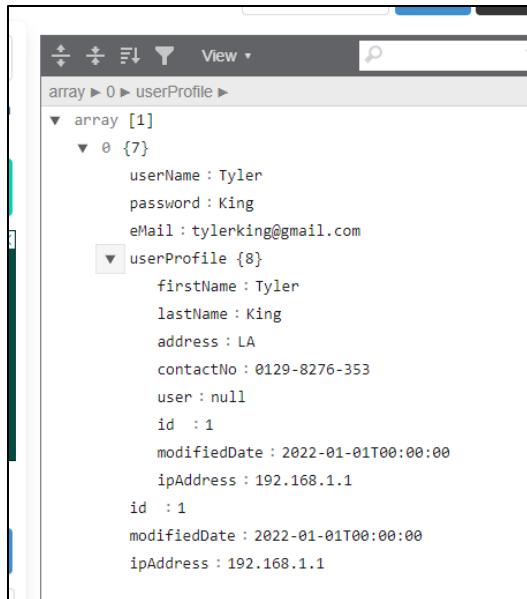


**Figure 10.14: Output of the Application**

9. Copy the JSON and place it within an online JSON Beautifier as shown in Figures 10.15 and 10.16.



**Figure 10.15: JSON Viewer**



**Figure 10.16: User Profile**

## 10.6 Security Considerations

Security consideration that should be taken in each layer are shown in Table 10.1.

Layer Name	Considerations
Domain Layer	<ul style="list-style-type: none"> <li>Input Validation: <ul style="list-style-type: none"> <li>Enforce strong validation within the domain entities and value objects to ensure that the data adheres to business rules.</li> <li>Use domain validation rules to reject invalid or malicious input at the earliest possible point.</li> </ul> </li> </ul>
Infrastructure Layer	<ul style="list-style-type: none"> <li>Data Validation: <ul style="list-style-type: none"> <li>Implement thorough input validation and sanitization when interacting with external data sources or services to prevent injection attacks.</li> <li>Use parameterized queries and stored procedures to mitigate SQL injection risks.</li> </ul> </li> </ul>
Service Layer	<ul style="list-style-type: none"> <li>Authentication: <ul style="list-style-type: none"> <li>Implement user authentication mechanisms in the service layer. Leverage ASP.NET Core's built-in authentication middleware or third-party authentication providers.</li> <li>Enforce strong password policies and consider multi-factor authentication for added security.</li> </ul> </li> </ul>
Onion Architecture Web API	<ul style="list-style-type: none"> <li>Input Validation: <ul style="list-style-type: none"> <li>Implement input validation at the API layer to sanitize and validate incoming requests.</li> <li>Leverage ASP.NET Core's model validation features to automatically validate incoming request models.</li> </ul> </li> </ul>

**Table 10.1: Security Considerations in Different Layers**

## 10.7 Monitoring and Tracing Setup for Error Detection and Performance Analysis

Setting up monitoring and tracing for error detection and performance analysis involves implementing tools and practices to gain insights into the operation of your application.

### 10.7.1 Monitoring for Error Detection

Here is a guide on how to set up monitoring and tracing:

#### Logging

Implement robust logging throughout your application code. Use a logging framework such as Serilog or log4net in ASP.NET Core.

#### Error Tracking Tools

Integrate error tracking tools such as Sentry, Raygun, or Application Insights.

#### Alerting

Set up alerting mechanisms to notify the operations team or developers when critical errors occur.

### 10.7.2 Monitoring for Performance Analysis

Now, let us delve into Monitoring for Performance Analysis to gain insights into optimizing application performance.

#### Instrumentation

Instrument your code to capture performance metrics. Use libraries such as the StatsD client or Application Insights SDK for ASP.NET Core.

#### Application Performance Monitoring (APM)

Implement APM tools such as New Relic, Datadog, or Azure Application Insights.

#### Custom Metrics

Define and measure custom metrics relevant to your application's specific performance goals.

### 10.7.3 Tracing Setup

After exploring Monitoring for Performance Analysis, the next step is to address Tracing Setup to enhance our understanding of request flows and system operations.

#### Distributed Tracing

Implement distributed tracing to trace the flow of requests across different services. Use tools such as Zipkin, Jaeger, or Application Insights.

#### Instrumentation Libraries

Integrate instrumentation libraries or middleware for tracing in your application code.

#### Correlation IDs

Use correlation IDs to link related log entries and traces across different components.

#### 10.7.4 Logging and Monitoring Infrastructure

Let us proceed to examine Logging and Monitoring Infrastructure to gain deeper insights into request flows and system operations.

##### Centralized Logging

Set up centralized logging infrastructure using tools such as Elasticsearch, Logstash, and Kibana (ELK Stack) or Azure Monitor Logs.

##### Metrics Storage and Visualization

Choose a metrics storage solution such as Prometheus or InfluxDB.

#### 10.7.5 Continuous Improvement

After having seen Logging and Monitoring Infrastructure, let us transition to exploring Continuous Improvement.

##### Performance Testing

Include performance testing as part of your Continuous Integration/Continuous Deployment (CI/CD) pipeline.

##### Regular Review and Optimization

Regularly review monitoring data and trace information to identify areas for improvement.

## Summary

- ✓ Four different layers of the Onion Architecture are domain entities layer, repository layer, service layer, and UI layer.
- ✓ For each of these layers, a corresponding project must be created to implement the Onion Architecture in an ASP.NET Core application.
- ✓ Domain entities layer contains the class library, POCO, and configuration classes. It also helps in creating database tables.
- ✓ The repository layer implements the interface for the generic repository class. It also contains the DbContext class.
- ✓ The service layer contains the business logic and user interfaces.
- ✓ The UI layer is the entry point of the program and the exterior layer of the Onion Architecture.
- ✓ In ASP.NET Core development with Onion Architecture, comprehensive security is very important.

# Test Your Knowledge



1. Which layer helps in creating database tables?

<b>A</b>	Domain entities layer
<b>B</b>	Repository layer
<b>C</b>	Service layer
<b>D</b>	UI layer

2. The repository layer contains the which of these classes?

<b>A</b>	UserRepository
<b>B</b>	UserProfile
<b>C</b>	DbContext
<b>D</b>	None of these

3. Under which folder is the User class created?

<b>A</b>	Entity Framework
<b>B</b>	Domain Entities
<b>C</b>	Repository
<b>D</b>	Models

4. What is the infrastructure layer also called?

<b>A</b>	Domain entities layer
<b>B</b>	Repository layer
<b>C</b>	Service layer
<b>D</b>	UI layer

5. Which is the exterior layer of the Onion Architecture?

<b>A</b>	Domain entities layer
<b>B</b>	Repository layer
<b>C</b>	Service layer
<b>D</b>	UI layer

## Answers

1	A
2	C
3	D
4	B
5	D

## **Try It Yourself**

1. Create a user service in the service layer responsible for user authentication. Implement a method for validating user credentials (username and password).
2. Create a domain entity for a user profile with relevant properties (example, username, email, and password). Implement input validation within the domain entity to ensure data integrity and adherence to business rules. Use attributes such as Required, StringLength, and custom validation attributes to enforce validation rules.
3. Set up data access using Entity Framework Core in the infrastructure layer. Implement thorough data validation and parameterization in database queries to prevent SQL injection attacks. Use EF Core's validation features and parameterized queries to mitigate injection risks.

# *Session 11: Overview of Fluent Model and AutoMapper in ASP.NET Core MVC*

## **Session Overview**

This session delves into Fluent API and Fluent model, explaining their various models and mappings. It also outlines the advantages of Fluent API and Fluent model. Additionally, the session provides a comprehensive procedure for executing operations such as Create, Edit, Delete, and Update (CRUD).

Further, the session covers AutoMapper and its effectiveness. The session elaborates on the reasons to use AutoMapper and provides guidance on its installation and configuration.

## **Objectives**

In this session, students will learn to:

- ✓ Explain Fluent API and Fluent model
- ✓ Describe benefits of Fluent API and Fluent Model
- ✓ Outline the procedure to perform operations such as Create, Edit, Delete, and Update (CRUD) using Fluent API with ASP.NET Core MVC
- ✓ Define AutoMapper
- ✓ List reasons for using AutoMapper
- ✓ Explain the installation and configuration process for AutoMapper
- ✓ Describe use of AutoMapper in an application
- ✓ Explain Minimal APIs

## **11.1 Introduction to Fluent API and Fluent Model**

Fluent APIs refer to APIs that are easy to read. They are based on the concept of method chaining. They allow configuring entities and their properties. Fluent APIs support both Entity mapping and Properties mapping.

Entity Framework (EF) code works by mapping Plain Old CLR Objects (POCO) classes to tables. POCO classes are not dependent on any base class that relies on a framework. POCO classes are mapped using certain conventions of EF. However, sometimes, there is additional requirement and conventions may not be followed. In such cases, EF Fluent APIs or Data Annotations are used. They offer advanced ways of model configuration. Fluent APIs provide additional configurations that may not be possible with Data Annotations. Fluent APIs and Data Annotations can work together but as per Code First approach, Fluent API is preferred because it provides more functionality than Data Annotations.

Therefore, when there are entities that cannot be mapped as per conventions, then the EF Fluent API is used. The `ModelBuilder` class in EF Core serves as a Fluent API.

Features of a model that EF Core Fluent API allows customizing are as follows:

### Model

Configures the database mappings for an EF model. The default schema, DB functions, additional data annotation properties, and entities to be omitted from mapping are all configured here.

### Entity

Entity mappings are simple mappings that will impact the understanding of EFs and how the classes are mapped to the databases.

Configures entity-to-table and relationship mappings, such as PrimaryKey, AlternateKey, index, table name, one-to-one, one-to-many, and many-to-many relationships, and so on.

### Property

Properties mapping is used to configure attributes for each property of an entity or complex type. It can also be used to obtain a configuration object for a given property. The properties of domain classes can also be mapped and configured using Fluent API.

Configures column name, default value, nullability, foreign key, data type, concurrency column, and other properties.

To access Fluent API, one should override the `OnModelCreating()` method in `DbContext`.

Code Snippet 1 shows an example of renaming the column name in the `Student` table from `FirstName` to `StudentFirstName` using this method. This is done in the file `StudentContext.cs`. Assume that an MVC application has been created and opened in Visual Studio 2022 IDE. Follow the steps given in Session 3 to add data handling functionality. The `OnModelCreating()` method for the Fluent API implementation is overridden here.

#### Code Snippet 1 (`StudentContext.cs`)

```
public class StudentContext : DbContext{
    public StudentContext(
        DbContextOptions<StudentContext>options) : base(options) {
    protected override void OnModelCreating(ModelBuilder
        modelBuilder) {
        modelBuilder.Entity<Student>().Property(e => e.FirstName)
            .HasColumnName("StudentFirstName");
    }
    public virtual DbSet<Course> Courses { get; set; }
```

```
public virtual DbSet<Enrollment> Enrollments {  
    get; set; }  
public virtual DbSet<Student> Students {  
    get; set; }  
}
```

In Code Snippet 1, `ModelBuilder` is a tool that helps map Common Language Runtime (CLR) classes to database schemas. This is the main class where all domain classes are set up. Fluent API provides several important methods and some of them are as follows:

### Entity<TEntityType>()

`DbModelBuilder.Entity<TEntityType>()` returns an object that can be used to configure an entity type that is registered as part of the model. For the same entity, this method can be called numerous times to complete multiple lines of configuration.

### HasKey<TKey>()

The primary key property for this entity type is configured.

## 11.2 Benefits of Fluent API

Fluent API is a complex approach for specifying model settings, which includes features beyond what Data Annotations can provide. There are several advantages of using Fluent API as a programmer. Some are as follows:

- A fundamental design of Fluent API leads to clear visibility
- Fluent APIs draw attention to the goal of the code
- Fluent APIs make it simple to manipulate complex notions
- It is highly flexible and everything that can be configured through Data Annotations is also possible through Fluent API

## 11.3 Working with Fluent API

The procedure to configure entities and properties using Fluent API is described through an example. In the example, a class called `Customer` is created with its `DbContext`. By using Fluent API, `OnModelCreating()` method in the `Context` class is redefined. `Controller` class is used to call the CRUD methods and operate with them. Steps involved in the process are as follows:

1. Create an ASP.NET Core MVC application using Visual Studio 2022.
2. Create a class in the **Models** folder. Code Snippet 2 shows a class `Customer` in the `Customer.cs` file, which defines details of a customer.

## Code Snippet 2 (Customer.cs)

```
public class Customer
{
    public int CustomerId { get; set; }
    public string CustomerName { get; set; }
    public string Address { get; set; }
}
```

3. Create the Context class for the Model. A context must be created in the EF so that it can interact with a table named Customer. Follow the steps given in Session 3 to add data handling functionality. As shown in Code Snippet 3 (code for the file CustomerContext.cs), CustomerContext is the class inherited from DbContext class to implement the EF functionality. Ensure that the connection string name given in appsettings.json match with the one given in Code Snippet 3.

## Code Snippet 3 (CustomerContext.cs)

```
Line 14 public class CustomerContext :DbContext {
Line 15     public CustomerContext() :
                base("name=DbConnectionString")
Line 16     {
Line 17     }
Line 18     public DbSet<Customer> Customers { get; set; }
Line 19     protected override void OnModelCreating(
                modelBuilder)
Line 20{
Line 21     modelBuilder.Entity<Customer>().HasKey(p =>
                p.CustomerId);
Line 22     modelBuilder.Entity<Customer>().Property(c =>
                c.CustomerId);
Line 23     base.OnModelCreating(modelBuilder);
Line 24     }
Line 25 }
```

The main explanation of the code is as follows:

- A constructor called CustomerContext connects with the base class.
- It provides a reference to the Customer class and returns DbSet of the Customer class.
- OnModelCreating() method carries Fluent API tools.

Table 11.1 explains some of the code statements from Code Snippet 3 in more detail.

Code Line	Description
Line 19	<ul style="list-style-type: none"><li>• The OnModelCreating method takes one parameter, which is a ModelBuilder object. This ModelBuilder</li></ul>

Code Line	Description
	<p>class is responsible for mapping POCO classes to database schema. When the first instance of a derived context is produced, this function is only called once.</p> <ul style="list-style-type: none"> <li>The model for that context is then cached and it will be used for all subsequent occurrences of the context in the app domain.</li> </ul>
Line 21	The HasKey() method can be used to explicitly set a property as a primary key.
Line 22	Property configures a struct property that is defined on this type.

**Table 11.1: Customer Context**

More methods such as HasRequired() or HasForeignKey() of Fluent API can be extended and used as shown in Code Snippet 4.

**Code Snippet 4** (CustomerContext.cs)

```
modelBuilder.Entity<SeniorCitizen>().HasRequired(p =>
    p.Customer)
    .WithMany(s =>
    s.SeniorCitizen).HasForeignKey(s=>s.CustomerId);
```

In Code Snippet 4, SeniorCitizen is another class created. Primary key and Foreign Key relations are established between Customer and SeniorCitizen classes. The controller class is configured to provide customer's specifications and implement CRUD operations.

The CustomerContext class specification has been taken within this controller so that controller relates to the model.

4. Create a Customer in the file CustomerController.cshtml as shown in Code Snippet 5. `HttpPost` based Create method (which takes Customer class reference) is defined so that when these values are found in view, it can be received and added to the customer via context.

**Code Snippet 5** (CustomerController.cshtml)

```
#region Create Customer
    public ActionResult Create()
    {
        return View(new Customer());
    }
    [HttpPost]
    public ActionResult Create(Customer Fresh)
    {
```

```

        objContext.Customers.Add(Fresh);
        objContext.SaveChanges();
        return RedirectToAction("Index");
    }
#endregion

```

5. Edit a Customer in the file CustomerController.cshtml as shown in Code Snippet 6. `HttpPost` based `Edit` method searches for the specific customer via `CustomerId` and then, makes changes in that object. Further changes in the database will be implemented by the `SaveChanges()` method of the `DbContext`.

#### Code Snippet 6 (CustomerController.cshtml)

```

#region edit Customer
    public ActionResult Edit(int id)
    {
        Customer Customer = objContext.Customers.Where(x
            => x.CustomerId == id).SingleOrDefault();
        return View(Customer);
    }
    [HttpPost]
    public ActionResult Edit(Customer model)
    {
        Customer Customer = objContext.Customers.Where(x
            => x.CustomerId ==
            model.CustomerId).SingleOrDefault();
        if (Customer != null)
        {
            objContext.Entry(
                Customer).CurrentValues.SetValues(model);
            objContext.SaveChanges();
            return RedirectToAction("Index");
        }
        return View(model);
    }
#endregion

```

6. Delete a Customer record through the file CustomerController.cshtml as shown in Code Snippet 7. The `Delete()` method of `HttpPost` gets the reference of a customer whose details must be deleted. When the object has been tracked, it is removed via the `Remove()` method of `DbContext` class.

#### Code Snippet 7 (CustomerController.cshtml)

```

#region Delete Customer
    public ActionResult Delete(int id)
    {
        Customer Customer =
            objContext.Customers.Find(id);
        return View(Customer);
    }

```

```

        }
        [HttpPost]
        public ActionResult Delete(int id, Customer model)
        {
            var Customer = objContext.Customers.Where(x =>
                x.CustomerId == id).SingleOrDefault();
            if (Customer != null)
            {
                objContext.Customers.Remove(Customer);
                objContext.SaveChanges();
            }
            return RedirectToAction("Index");
        }
    #endregion

```

The `Delete` method of `HttpPost` retrieves the reference of a customer, whose details must be deleted. When the object has been tracked, it is removed via the `Remove` method of `DbContext` class.

7. Fetch a `Customer` as shown in Code Snippet 8. The `Details` method fetches the record that matches the `CustomerId` in contrast to the `id` passed to this method.

#### Code Snippet 8 (`CustomerController.cshtml`)

```

#region List and Details Customer
    public ActionResult Index()
    {
        var Customers = objContext.Customers.ToList();
        return View(Customers);
    }
    public ViewResult Details(int id)
    {
        Customer Customer =
            objContext.Customers.Where(
                x=>x.CustomerId==id).SingleOrDefault();
        return View(Customer);
    }
#endregion

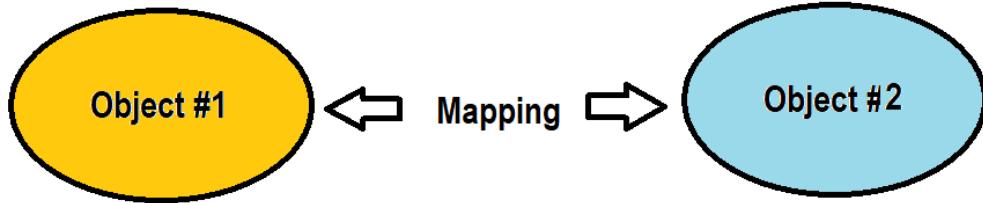
```

Build and run the application. In this manner, one can implement CRUD operations through Fluent API in ASP.NET Core MVC applications.

## 11.4 Introduction to AutoMapper

AutoMapper is a small library for .NET that allows changing the type of an object. It is a convention-based object-to-object mapper, which requires minimal changes. Object-to-object mapping, as shown in Figure 11.1, is required when there are a large number of members in objects. AutoMapper changes an input object of a

particular type to an output object that is different. AutoMapper also saves time and effort while manually mapping the properties of incompatible types in applications.



**Figure 11.1: Object-to-Object Mapping**

Any set of classes can be mapped with AutoMapper, but the properties of the classes must have the same names. If the property names do not match, then manually update the code.

## 11.5 Uses of AutoMapper

AutoMapper can be useful in following scenarios:

- Objects of similar or distinct types are to be mapped.
- Application models, also known as Entities, correspond to the database tables of the database in use.

For example, for a model class called `Movies`, each field in the database of the `Movies` table will have corresponding properties. In the display layer, all fields of the `Movies` class may not be required. Only three properties may be required in the `Movies` entity in the presentation layer, such as `MovieCode`, `MovieName`, and `MovieBudget`.

When sending data from the business layer to the presentation layer, additional characteristics that do not match the models established in the application may be required. For example, a few properties, such as `ReorderLevel`, `ReorderValue`, and `NetPrice`, which are not available in the `Product` class, are required in the presentation layer for the product entity.

Model class attributes are different from data transfer object class properties. To synchronize these two, a large number of classes or methods that match those properties must be provided and instances must be converted to other types. However, when the application is built on the Enterprise Application Platform (EAP), it is tedious to manually map a large number of model entities.

To automatically map entities to model, AutoMapper is used. AutoMapper cannot only be used in mapping similar objects, but can also be used for dissimilar objects. Hence, it can be used in mapping entities that interact with the database with entities that interact with a client.

For example, consider two classes, `Movies` and `MoviesDTO` as shown in Code Snippet 9, which must be mapped.

### Code Snippet 9

```
public class Movies
{
    public string MovieCode{ get; set; }
    public string MovieName{ get; set; }
    public int MovieBudget { get; set; }

}
public class MoviesDTO
{
    public string MovieCode{ get; set; }
    public string MovieName{ get; set; }
    public int MovieBudget { get; set; }
}
```

Code Snippet 10 shows how to build a method that takes a `MoviesDTO` class object as an input, maps the data, and returns a `Movie` class object.

### Code Snippet 10

```
public Movies MapObjects(MoviesDTO mdto)
{
    return new Movies()
    {
        MovieName = mdto.MovieName,
        MoviewName= mdto.MoviewName,
        MovieBudget = mdto.MovieBudget
    };
}
```

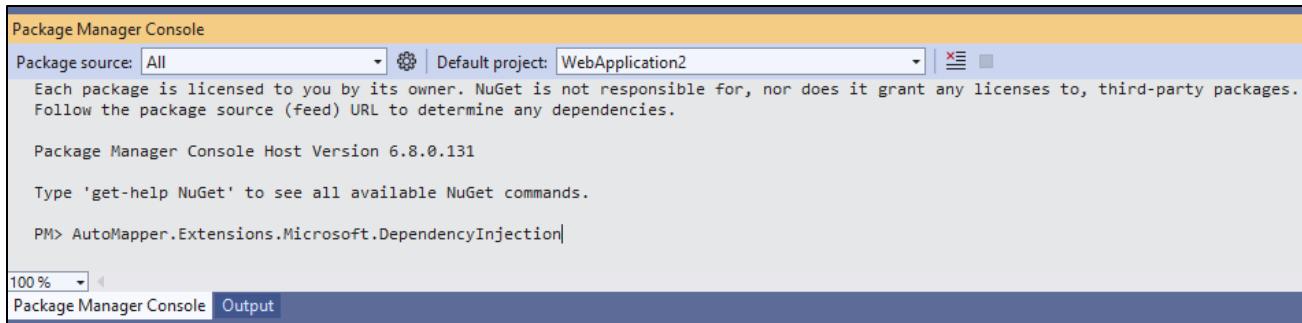
Though this solves the problem, this cannot be replicated when there are several classes with many properties. In all these cases, AutoMapper will help.

## 11.6 Working with AutoMapper

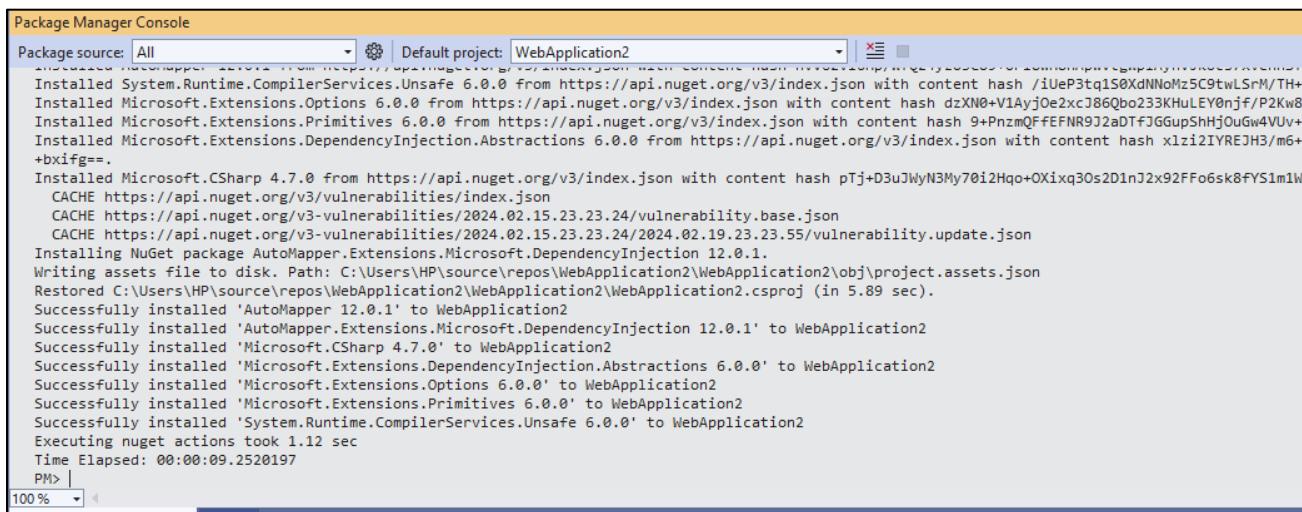
Reflection, which has been used in traditional C# programming, provides metadata information about Assemblies, Types, Methods, and so on. Reflection can call the methods within a class of a given assembly programmatically.

AutoMapper also uses the same technique. Following steps describe how to use AutoMapper with Web API from .NET Core:

1. Install `AutoMapper.Extensions.Microsoft.DependencyInjection` package through **Package Manager Console** as shown in Figures 11.2 and 11.3.



**Figure 11.2: Process of Installing AutoMapper**



**Figure 11.3: Installed AutoMapper**

2. Open the `Program.cs` file in your ASP.NET Core project. Add the `using` directive for AutoMapper. Adding the `using AutoMapper;` directive at the top of the `Program.cs` file allows you to use AutoMapper classes and methods within that file and throughout your project. Add the line `builder.Services.AddAutoMapper(typeof(Program).Assembly);` to configure AutoMapper. This line tells ASP.NET Core to configure AutoMapper and scan the assembly where the `Program` class resides for classes inherited from AutoMapper's profile class as shown in Code Snippet 11.

### Code Snippet 11

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using AutoMapper;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
```

```

builder.Services.AddControllers();

// Add AutoMapper configuration
builder.Services.AddAutoMapper(typeof(Program).Assembly);

// Learn more about configuring Swagger/OpenAPI at
// https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();

```

3. Next, create the `Movies` class as shown in Code Snippet 12.

### Code Snippet 12

```

namespace WebApplication2
{
    public class Movies
    {
        public string MovieCode { get; set; }
        public string MovieName { get; set; }
        public int MovieBudget { get; set; }
    }
}

```

4. Similarly, create `MoviesDTO` class. Code Snippet 13 shows how to create a `MoviesDTO` class for data transformation.

### Code Snippet 13

```

namespace WebApplication2
{
    public class MoviesDTO
    {
        public string MovieCode { get; set; }
        public string MovieName { get; set; }
        public int MovieBudget { get; set; }
    }
}

```

```
    }  
}
```

5. As shown in Code Snippet 14, create `AutoMapperProfile` class that inherits from `Profile` where the mapping between `Movies` and `MoviesDTO` objects happens.

#### Code Snippet 14

```
using AutoMapper;  
  
namespace WebApplication2  
{  
    public class AutoMapperProfile : Profile  
    {  
        public AutoMapperProfile()  
        {  
            CreateMap<MoviesDTO, Movies>();  
        }  
    }  
}
```

`MovieDTO` has now been mapped to the `Movies` class. When the program starts, `AutoMapper` is initialized and it scans all assemblies for classes that are inherited from the `Profile` class, loading their mapping configurations.

6. Using the code in Code Snippet 15, map the two items using the `IMapper` interface. Here, `MoviesController.cs` is the name of the new controller. In the controller function `Object() { [native code] }`, resolve the `IMapper` requirement as shown in Code Snippet 15.

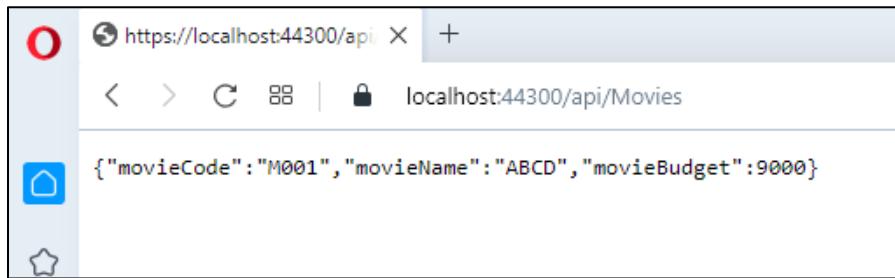
#### Code Snippet 15

```
using Microsoft.AspNetCore.Mvc;  
using AutoMapper;  
using WebApplication2;  
  
[ApiController]  
[Route("api/[controller]")]
public class MoviesController : ControllerBase  
{  
    private readonly IMapper _mapper;  
  
    public MoviesController(IMapper mapper)  
    {  
        _mapper = mapper;  
    }  
  
    [HttpGet]
    public IActionResult Get()  
    {  
        MoviesDTO mDTO = new MoviesDTO()  
    }  
}
```

```
{  
    MovieCode = "M001",  
    MovieName = "ABCD",  
    MovieBudget = 9000  
};  
return Ok(_mapper.Map<Movies>(mDTO));  
}  
}
```

In the `Program.cs` file using the minimal API syntax, we directly instantiate a `MoviesDTO` object with sample values. Then, we retrieve the `IMapper` service from the application's services using `app.Services.GetRequiredService<IMapper>()`, and use the `Map` function to map the `MoviesDTO` object to the `Movies` class.

7. Build and launch the API in a browser. Then, go to `/api/Movies` to see the result as shown in Figure 11.4.



**Figure 11.4: Output of Using AutoMapper**

## 11.7 Minimal APIs

Minimal APIs are a lightweight approach for building HTTP services in ASP.NET Core. They offer a simpler, more streamlined alternative to traditional MVC controllers, enabling developers to quickly create APIs with minimal overhead. With Minimal APIs, developers can define endpoints and routes using a concise syntax, reducing boilerplate code and improving readability.

## Benefits of Minimal APIs

Following are some benefits of Minimal APIs:

### Simplicity

Minimal APIs simplify the process of creating HTTP services by eliminating the requirement of controllers and action methods. Developers can define endpoints and routes using a concise syntax, resulting in cleaner and more readable code.

### Reduced Boilerplate

Unlike traditional MVC controllers, which require the definition of controller classes and action methods, Minimal APIs allow developers to define routes and handle requests directly within a single file. This reduces boilerplate code and improves developer productivity.

### Faster Development

By streamlining the development process, Minimal APIs enable developers to build APIs more quickly and efficiently. With fewer abstractions and less configuration, developers can focus on writing business logic and delivering value to end users.

# Summary

- ✓ The Entity Framework (EF) Fluent API is used to override conventions in domain classes.
- ✓ The EF Core Fluent API allows customizing Model, Entity, and Property configuration.
- ✓ `DbModelBuilder` is a tool that helps to map CLR classes to database schemas.
- ✓ The main types of mapping that Fluent API supports are entity and Properties mapping.
- ✓ Fluent APIs make it simple to manipulate complex notions.
- ✓ The design of Fluent API provides clear visibility and makes it simple to manipulate notions.
- ✓ AutoMapper is a convention-based object-to-object mapper for .NET, which can be used when there are a large number of members in objects.
- ✓ AutoMapper changes an input object to an output object of a different type.
- ✓ AutoMapper saves time and effort while manually mapping the properties of incompatible types in the application.
- ✓ AutoMapper can be used not only in mapping similar objects, but also dissimilar objects.
- ✓ One can install AutoMapper in a project by adding `AutoMapper.Extensions.Microsoft.DependencyInjection` via Package Manager.
- ✓ Minimal APIs are a lightweight approach for building HTTP services in ASP.NET Core

# Test Your Knowledge



1. Where are the database mappings, the default schema, DB functions, additional data annotation properties, and entities to be omitted from mapping configured?

<b>A</b>	CRUD
<b>B</b>	Model
<b>C</b>	Entity
<b>D</b>	Properties

2. \_\_\_\_\_ is a tool that helps to map CLR classes to database schemas. It is the main class, where all domain classes are set up.

<b>A</b>	ModelBuilder
<b>B</b>	Context
<b>C</b>	Context
<b>D</b>	Customer

3. Which mapping is used to configure attributes for each property and map the properties of your domain classes using Fluent API?

<b>A</b>	CLR
<b>B</b>	Model
<b>C</b>	Entity
<b>D</b>	Properties

4. Which class is used to provide the specifications for Customer to implement CRUD operations?

<b>A</b>	Controller
<b>B</b>	Customer
<b>C</b>	CRUD
<b>D</b>	Create

5. Which method can be used to fetch the record that matches the CustomerId in contrast to the id passed to this method?

<b>A</b>	Action
<b>B</b>	DbContext
<b>C</b>	Details
<b>D</b>	Controller

6. \_\_\_\_\_ can help to call the methods within a class of a given assembly programmatically.

<b>A</b>	Reflection
<b>B</b>	AutoMapper
<b>C</b>	IMapper
<b>D</b>	IEnumerable

7. Any set of classes can be mapped with AutoMapper, but the properties of the classes must have \_\_\_\_\_ names.

<b>A</b>	Few
<b>B</b>	Many
<b>C</b>	Same
<b>D</b>	Different

8. What does AutoMapper use to retrieve a type of existing object and invoke its methods or access its fields and attributes?

<b>A</b>	IMapper
<b>B</b>	Reflection
<b>C</b>	_mapper.Map()
<b>D</b>	None of these

9. Which of the following is used to map two items during mapping in an ASP.NET application?

<b>A</b>	IMapper
<b>B</b>	Reflection
<b>C</b>	AutoMapper
<b>D</b>	None of these

## Answers

1	B
2	A
3	D
4	A
5	C
6	A
7	C
8	B
9	A

## **Try It Yourself**

1. You must create a new ASP.NET Core MVC project for an online shopping store. Define a simple model class, such as Product, with properties such as ProductId, ProductName, and Price. Use Fluent API to configure the primary key and property types in the `OnModelCreating` method of your `DbContext`. Finally, test the configuration by performing CRUD database operations using the configured model. Make use of Visual Studio 2022 and ASP.NET MVC and Core to do this.
2. Follow the steps and perform following tasks in Visual Studio 2022:
  - I. Add AutoMapper to an existing or new ASP.NET Core project using NuGet.
  - II. Create two classes, Order and OrderDTO, with similar and distinct properties.
  - III. Configure AutoMapper in Program.cs.
  - IV. Implement a controller method that uses AutoMapper to map an instance of OrderDTO to Order and return the result.
  - V. Test the mapping by invoking the controller method and inspecting the results.

# Session 12: .NET Core Token Authentication in Web Applications

## Session Overview

This session is meticulously crafted to empower students with a practical and in-depth understanding of token-based authentication within the context of ASP.NET Core. The session covers key aspects, from describing fundamentals of token-based authentication to hands-on implementation in MVC/Web Applications.

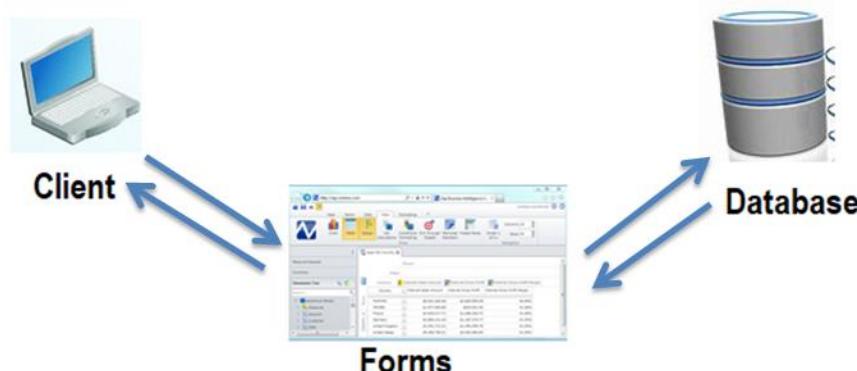
## Objectives

In this session, students will learn to:

- ✓ Describe token-based authentication
- ✓ Explain the process to validate tokens in ASP.NET Core
- ✓ Explain JSON Web Token
- ✓ Identify how to set up Token Authentication in .NET Core
- ✓ Elaborate on Token Generation and Issuing
- ✓ Explain how to use tokens obtained from identity providers in applications
- ✓ Describe Token-Based Authentication in MVC/Web Application

## 12.1 Overview

While developing any Web application, security considerations form an integral part of the development process. Earlier, Web application security was managed either by storing data in a session or using .NET forms-based authentication as represented in Figure 12.1.



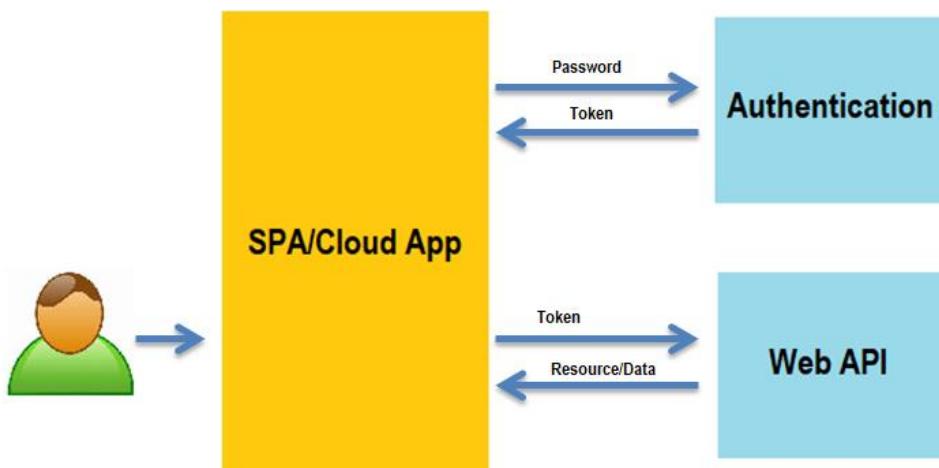
**Figure 12.1: Storing Data in Sessions**

However, storing data in a session led to a lot of clutter in data and other issues. This was handled by the introduction of cloud servers. The cloud servers can be brought

up and down as required. Hence, the session storage data is transferred to the cloud. The most preferred way of transferring session storage to the cloud is by using the Representational State Transfer (REST) design.

The cloud-based environment and Single Page Applications (SPAs) frequently visit a REST Web API or .NET MVC controller to obtain or send data as represented in Figure 12.2. This ensures transfer of business logic from the server to the client.

The cloud-based environment allows users to use Web service APIs directly from the user interface. Additionally, token-based authentication is also used.



**Figure 12.2: Data Storage in Could-Based Environment and SPAs**

Let us now understand token-based authentication.

## 12.2 Token-Based Authentication

Authentication refers to verification of a user identity before providing access to specific applications, modules, files, or data. Token-based authentication provides an additional encrypted security process in the authentication process. In a token-based authentication, after a user logs in using credentials, an access token is sent for secondary authentication. This token is valid for a particular period. Access to the application is valid till the time token is valid and may also expire if a user logs out. If the token code is invalid, then an error message appears.

Token-based authentication is very secure and difficult to be hacked. This kind of authentication is used with APIs in mobile applications and SPAs.

Token-based authentication can be performed through JSON Web Token (JWT). JWT is an open-source technology and can be used with any backend technology, such as ASP.NET, Python, or Java enterprise technologies. The transmission of information in JWT can be through a URL - in the HTTP header or POST parameter.

The format of the information is JSON and it has three sections, header, payload, and signature. The header section has metadata and algorithm for encryption. Payload contains the information that is transmitted. Signature verifies the data integrity. Code Snippet 1 shows a JWT creation in ASP.NET MVC Web API. It is assumed that this code is added inside Program.cs.

### Code Snippet 1

```
var securedkey = new  
    SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes("This  
is the secured Key"));  
var securedcredentials = new SigningCredentials(securedkey,  
    SecurityAlgorithms.HmacSha256);  
var claims = new[] {  
    new Claim(ClaimTypes.Name, usr.username),  
    new Claim(JwtRegisteredClaimNames.Sub,  
        usr.username),  
    new Claim(JwtRegisteredClaimNames.Jti,  
        Guid.NewGuid().ToString("N"))  
};  
var token = new JwtSecurityToken(issuer: "abcd.com",  
    audience: "abcd.com", claims: claims, expires:  
    DateTime.Now.AddMinutes(60), signingCredentials:  
    securedcredentials);  
return new JwtSecurityTokenHandler().WriteToken(token);  
. . .
```

Code Snippet 1 is designed to generate a JWT for user authentication. It begins by creating a symmetric key from the UTF-8 encoded byte representation of the string 'This is the secured Key'. This key is then used to create signing credentials, specifying the HMAC-SHA-256 algorithm for securing the JWT.

The code proceeds to define claims, including the user's name, a subject claim containing the username, and a unique JWT ID claim. Subsequently, a new JWT is instantiated with specified parameters such as issuer, audience, expiration time (set to 60 minutes from the current time), and the previously created signing credentials.

Finally, the code serializes the JWT into a string using JwtSecurityTokenHandler and returns the resulting token.

Some of the classes used in Code Snippet 1 are explained in Table 12.1.

Class	Description
JwtSecurityToken	A Security token designed for representing a JWT.
Claims	Claims refer to information and is a part of the payload . For example, an ID Token has a Claim Name that can hold the username.

Class	Description
JwtSecurityTokenHandler	A SecurityTokenHandler designed for creating and validating JWT.
JwtRegisteredClaimNames	It is a structure that provides names for public standardized claims.

Table 12.1: Classes Used in the Code

## 12.3 Setting up Token Authentication in .NET Core

Token authentication is a critical aspect of securing modern Web applications, providing a streamlined and secure method for user verification. In the context of .NET Core, configuring token authentication involves several steps to ensure a robust and reliable authentication mechanism. Here is a guide on how to set up token authentication in an ASP.NET Core application.

1. Open Visual Studio 2022.
2. Create a new ASP.NET Core Web API project.
3. Name the project suitably and click **Create**.
4. Install Required Packages:
  - Begin by installing the necessary NuGet packages for handling token authentication. The primary package is `Microsoft.AspNetCore.Authentication.JwtBearer`. Use the command in the Package Manager Console as follows:

```
Install-Package
Microsoft.AspNetCore.Authentication.JwtBearer
```

Figure 12.3 depicts this.

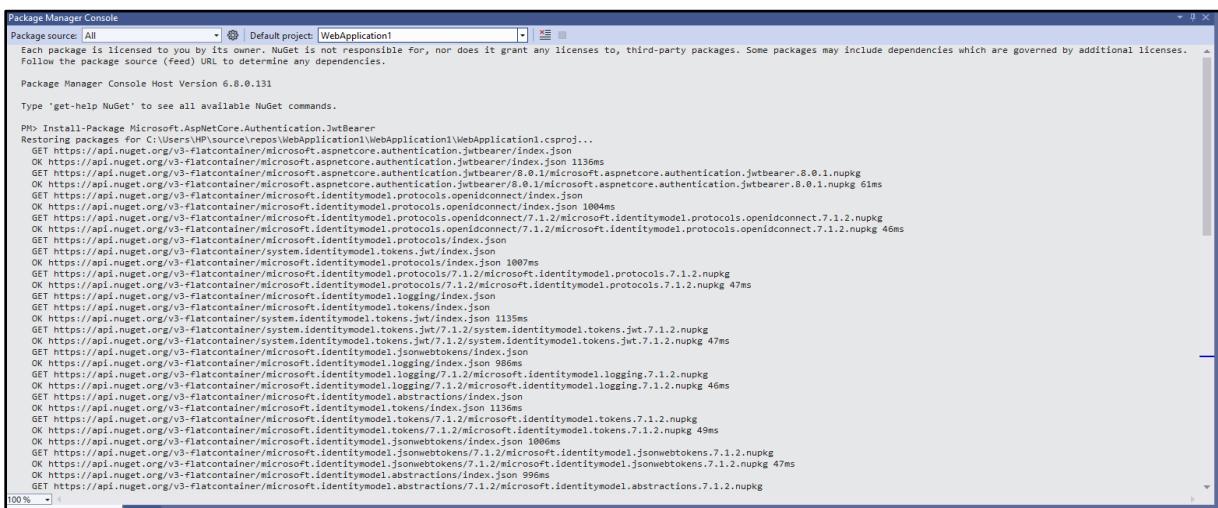


Figure 12.3: Installation of Required Packages

5. Configure JWT Authentication in `Program.cs`.

- Navigate to the Program.cs file and configure JWT authentication as shown in Code Snippet 2.

### Code Snippet 2

```

using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
{
    options.TokenValidationParameters = new
TokenValidationParameters
    {
        // Replace myapi with your actual API URL or issuer
        // Replace myclientapp with your actual client or audience
        // Replace mysecretkey123 with your actual secret key
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "https://myapi.com",
        ValidAudience = "myclientapp",
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("mysecretkey123
!"))
    };
});
});

// Add Swagger/OpenAPI services
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment()){
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseRouting();
// Add authentication and authorization middleware
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
app.Run();

```

Here is a breakdown of what each part does in Code Snippet 2:

- I. AddControllers:
    - This method adds controllers to the application's services.
    - It enables MVC pattern for handling HTTP requests.
  - II. AddAuthentication:
    - This method configures the authentication services.
    - It sets the default authentication scheme for both authentication and challenge to JWT Bearer authentication.
  - III. AddJwtBearer:
    - This method adds JWT Bearer authentication to the application's authentication pipeline.
    - It configures the validation parameters for JWT tokens, such as issuer, audience, lifetime, and the signing key.
  - IV. TokenValidationParameters:
    - These parameters specify how the received JWT token should be validated.
    - In this example, validation includes checking the issuer, audience, lifetime, and the signature using a symmetric security key.
6. Add Authentication Middleware.

In the `Configure` method, add statements to incorporate authentication middleware as shown in Code Snippet 3.

### Code Snippet 3

```
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
    // Other configurations

    app.UseAuthentication();
    app.UseAuthorization();

    // Other middleware configurations
}
```

## 7. Secure Your Endpoints.

To protect specific endpoints or controllers with token authentication, use the `[Authorize]` attribute. An example is shown in Code Snippet 4.

## Code Snippet 4

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class SecureController : ControllerBase {
    [HttpGet]
    public IActionResult GetSecureData() {
        // Your secured endpoint logic
        return Ok("This is secure data accessible only with a
valid token.");
    }

    [HttpPost]
    public IActionResult PostSecureData([FromBody]
SecureDataModel data)
    {
        // Your secured endpoint logic
        // Process the secure data
        return Ok($"Secure data received and processed. Id:
{data.Id}");
    }
}
```

Figure 12.4 depicts this in the IDE.

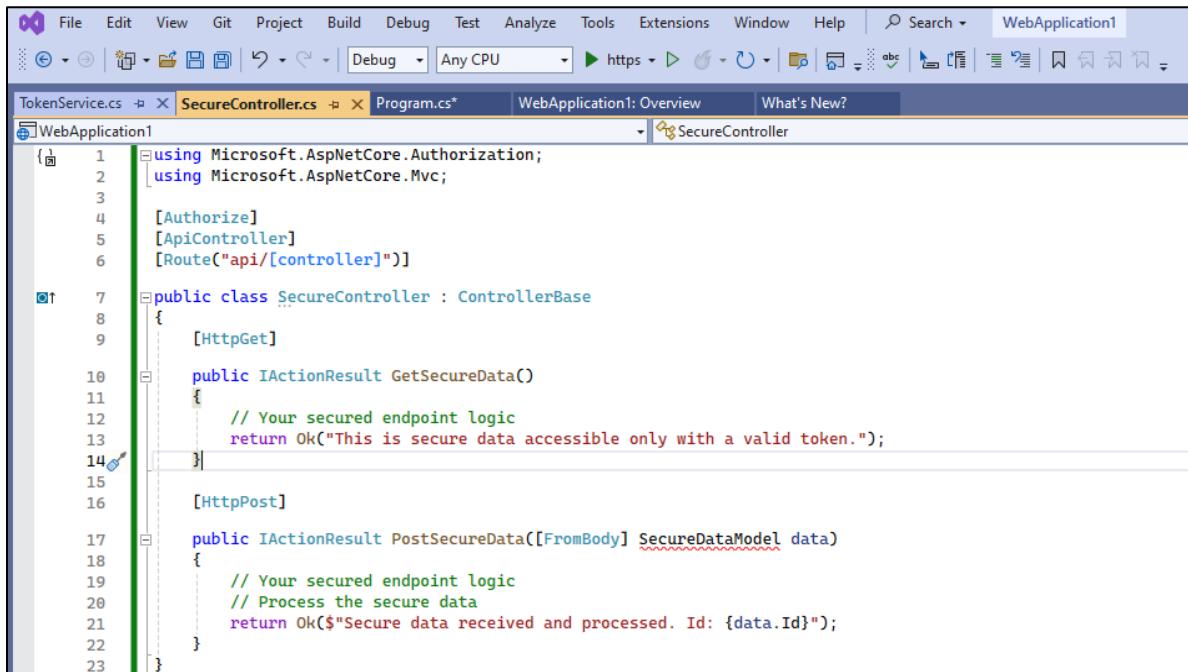


Figure 12.4: Securing Endpoints

8. Generate and Issue Tokens.

Code Snippet 5 is an example of how developers can implement a basic token service to generate and issue tokens in a .NET Core application. This example uses the `System.IdentityModel.Tokens.Jwt` library for working with JWTs.

## Code Snippet 5

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using Microsoft.IdentityModel.Tokens;

public class TokenService
{
    private readonly string _secretKey;
    private readonly string _issuer;
    private readonly string _audience;

    public TokenService(string secretKey, string issuer,
string audience)
    {
        _secretKey = secretKey;
        _issuer = issuer;
        _audience = audience;
    }

    public string GenerateToken(string userId, string
userName, string[] roles)
    {
        var claims = new[]
        {
            new Claim(ClaimTypes.NameIdentifier, userId),
            new Claim(ClaimTypes.Name, userName),
            // Include additional claims as required
            new Claim(ClaimTypes.Role, string.Join(",", roles))
        };

        var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_secretKey));
        var credentials = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(
            issuer: _issuer,
            audience: _audience,
            claims: claims,
            expires: DateTime.UtcNow.AddHours(1), // Token
expiration time
            signingCredentials: credentials
        );
    }
}
```

```

    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
}

```

Figure 12.5 depicts this in the IDE.

```

1  using System;
2  using System.IdentityModel.Tokens.Jwt;
3  using System.Security.Claims;
4  using System.Text;
5  using Microsoft.IdentityModel.Tokens;
6
7  public class TokenService
8  {
9      private readonly string _secretKey;
10     private readonly string _issuer;
11     private readonly string _audience;
12
13    public TokenService(string secretKey, string issuer, string audience)
14    {
15        _secretKey = secretKey;
16        _issuer = issuer;
17        _audience = audience;
18    }
19
20    public string GenerateToken(string userId, string userName, string[] roles)
21    {
22        var claims = new[]
23        {
24            new Claim(ClaimTypes.NameIdentifier, userId),
25            new Claim(ClaimTypes.Name, userName),
26            new Claim(ClaimTypes.Role, string.Join(", ", roles))
27        };
28
29        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_secretKey));
30        var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
31
32        var token = new JwtSecurityToken(
33            issuer: _issuer,
34            audience: _audience,
35

```

**Figure 12.5: Generating and Issuing Tokens**

## 9. Test the Authentication.

Use tools such as Postman or curl to test token authentication. Include the token in the Authorization header of requests as shown in Code Snippet 6.

### Code Snippet 6

Authorization: Bearer <your\_generated\_token>

In order to view output, one should have a working API and then, URL of the application's authentication server will be pasted in valid issuer field. Alternatively, if a fully developed application is hosted on third-party authentication provider such as Azure AD or Auth0, one could use their issuer URL. Using third party authentication will require paid subscription and a fully developed application. In practical real-world scenarios, this would be necessary.

## 12.4 Validating Tokens in ASP.NET Core

The tokens must be validated after it has been created. Tokens can be validated only as a string as it is transferred using a header such as x-api-token.

### 12.4.1 Automatic Authorization of Metadata

Details required to validate the tokens are available in metadata. This metadata can be retrieved from the authorization server using JwtBearer middleware. The metadata is retrieved only the first time. After JwtBearer middleware retrieves the data, it is configured.

To validate a token using JwtBearer, create and open a new ASP.NET Core Web API project in Visual Studio 2022.

Following are steps to validate a token using JwtBearer:

1. Add a reference to JwtBearer using

Microsoft.AspNetCore.Authentication.JwtBearer package. You can do this using the Package Manager Console or the .NET CLI.

2. Configure Authentication in Program.cs:

In the Program.cs file, configure authentication services and middleware directly. This involves setting up JwtBearer as the default authentication scheme and configuring JwtBearer options as shown in Code Snippet 7.

#### Code Snippet 7

```
var builder = WebApplication.CreateBuilder(args);
// Configure services
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
{
    // Configure JwtBearer options (for example, Authority,
    Audience, or TokenValidationParameters)
});

// Additional service configurations...

var app = builder.Build();

// Configure middleware
app.UseAuthentication(); // Enable authentication middleware

// Additional middleware and configuration...

app.Run();
```

3. Configure Middleware in `Program.cs`. Middleware configuration is done directly in the `Program.cs` file as shown in Code Snippet 8.

### Code Snippet 8

```
var app = builder.Build();

// Configure middleware
if (builder.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseAuthentication(); // Enable authentication middleware
app.UseStaticFiles();

// Additional middleware and configuration...

app.Run();
```

In this configuration process, steps outlined in the Code Snippets 7 and 8 aim to integrate `JwtBearer` authentication into a .NET Core application using the simplified hosting model introduced in .NET 6 onwards.

Initially, the application is equipped with the necessary tools by adding a reference to the `Microsoft.AspNetCore.Authentication.JwtBearer` package. Moving to the `Program.cs` file, the authentication services are configured with `AddAuthentication`, designating `JwtBearer` as the default authentication scheme.

The subsequent `AddJwtBearer` method facilitates further customization of `JwtBearer` options, allowing developers to specify parameters such as authority, audience, and token validation settings. Subsequently, in the middleware configuration section, the `UseAuthentication` method enables the `JwtBearer` authentication middleware, responsible for validating incoming tokens.

#### 12.4.2 Specifying Token Validation Parameters

The parameters for token validation can also be manually added as shown in Code Snippet 9. `Issuer`, `audience`, `signingkey`, `dateissuer`, and `clock skew` are the most common token validation parameters that are used. Depending on whether a symmetric or asymmetric key is used, the keys to be used to sign in the token are be submitted.

### Code Snippet 9

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
```

```

    {
        options.TokenValidationParameters = new
            TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            ValidateIssuer = true,
            ValidIssuer = "abcd.com",
            ValidateAudience = true,
            ValidAudience = "abcd.com",
            ValidateLifetime = true,
            IssuerSigningKey = new
                SymmetricSecurityKey(System.Text.Encoding.UTF8.Get
                    Bytes("This is the secured Key"))
            );
        });
    }
}

```

Some parameters that are used for token validation are as follows:

- **ValidateAudience**: This is used to validate the recipient of the token.
- **ValidAudience**: It is the value of the audience that is used for validation.
- **RequireExpirationTime**: This is used to check if the token has any expiry time.
- **ValidateLifetime**: This is used to check expiry of the token.
- **ClockSkew**: This is used to validate time.
- **IssuerSigningKeys**: This is used to validate signature of the token.
- **ValidateIssuer**: This is used to validate the issuer of the token, the server that generates the token.

## 12.5 More About JWT

JWT serves as a pivotal element in the realm of token-based authentication, playing a crucial role in securing and transmitting information between parties. In the context of .NET Core Token Authentication, understanding JWT is fundamental.

### 12.5.1 Definition and Structure

JWT is a compact, self-contained means of transmitting information between two parties. It is often used for authentication and data exchange in Web development.

### 12.5.2 Components of a JWT

JWTs consist of three main parts separated by dots: Header, Payload, and Signature. These components encode information in a base64 format, forming a secure and concise representation.

They are explained are as follows:

**Header**: Contains metadata about the type of token and the signing algorithm used. It is shown in Code Snippet 10.

## Code Snippet 10

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

**Payload:** Carries the claims. Claims are statements about an entity (typically the user) and additional data. It is shown in Code Snippet 11.

## Code Snippet 11

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "exp": 1516239022  
}
```

**Signature:** Created by encoding the header, payload, and a secret key using the specified algorithm. It ensures the integrity of the token and validates its source.

### 12.5.3 Use Cases in .NET Core

In the context of .NET Core, JWTs find extensive application in two primary scenarios:

#### Authentication:

- JWTs are widely used for user authentication. Once a user logs in, a JWT is issued containing relevant claims such as user ID and roles. This token is then sent with subsequent requests to authenticate the user.

#### Authorization:

- The claims within a JWT can be used to enforce access control policies, allowing or restricting users' access to specific resources based on their roles or permissions.

### 12.5.4 Integration in .NET Core Token Authentication

.NET Core provides middleware for handling JWTs. Developers can use the `Microsoft.AspNetCore.Authentication.JwtBearer` package to configure JWT authentication in the application.

.NET Core enables seamless validation of JWTs, ensuring that the received token is valid, not expired, and signed by a trusted authority.

### 12.5.5 Security Considerations

Following are key security considerations when using JWTs:

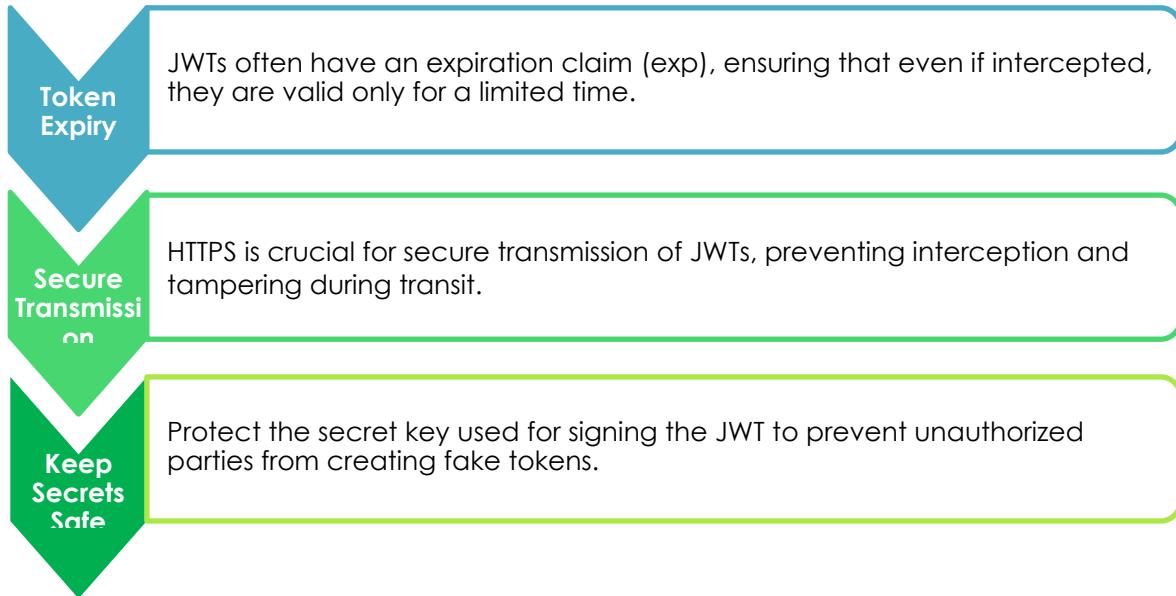


Table 12.2 provides a clear comparison between JWT and their integration in .NET Core Token Authentication, covering aspects such as structure or use cases.

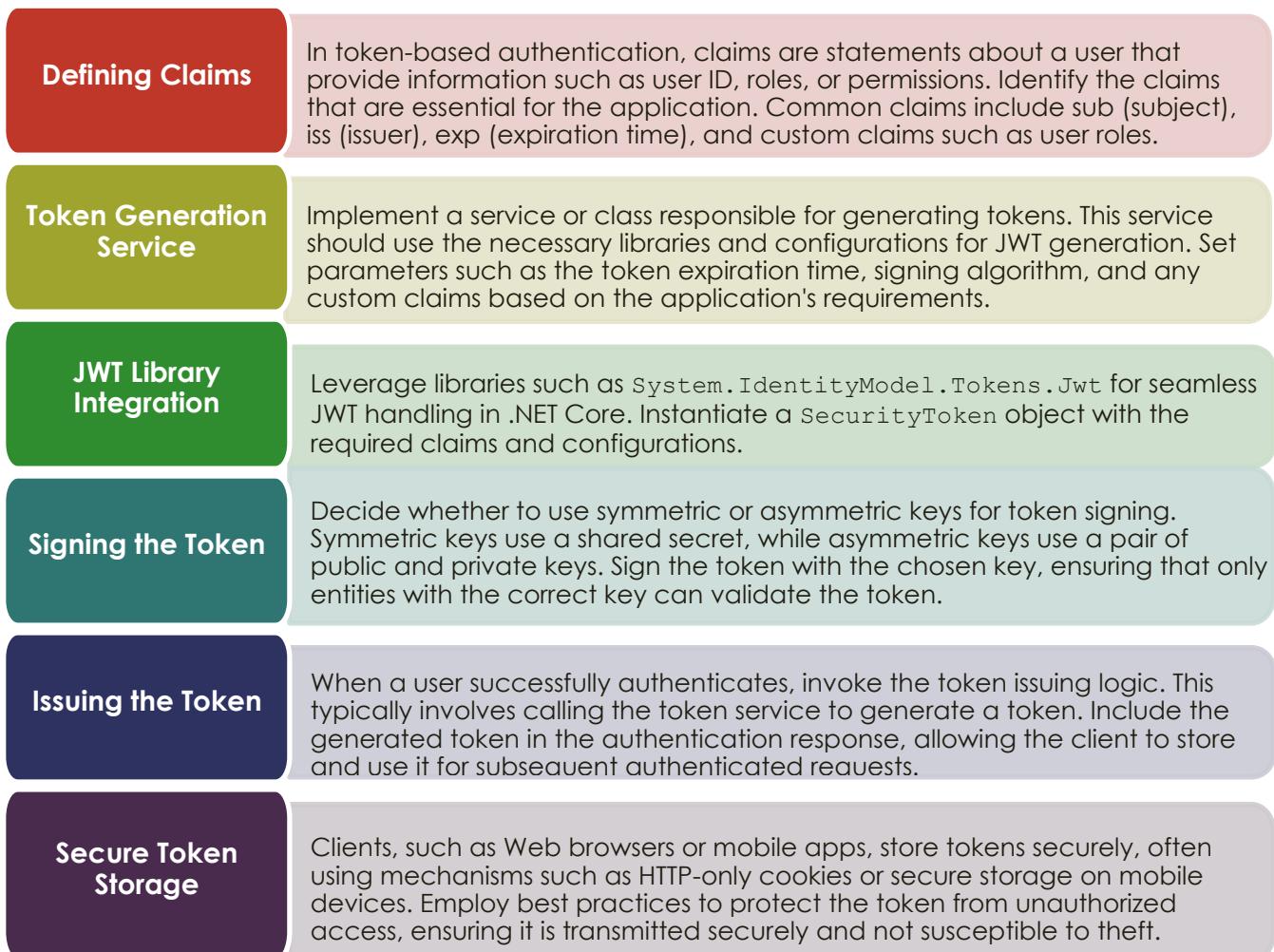
Aspect	JWT	.NET Core Token Authentication
Definition and Purpose	Compact means of transmitting information	Framework for secure user authentication
Structure	Header, Payload, Signature	Middleware, Token Validation
Header	Metadata about token and algorithm	Configured in Middleware
Payload	Carries claims about the user	Carries user information and additional data
Signature	Ensures token integrity	Validates token source
Use Cases in .NET Core	User Authentication, Authorization	Enforces access control policies based on claims

**Table 12.2: Comparison Between JWT and .NET Core Token Authentication**

### 12.6 Token Generation and Issuing

Token generation and issuing are integral components of a secure authentication system in .NET Core applications. These processes involve creating a token with relevant user information and cryptographic signatures, ensuring the token's integrity and authenticity.

The process involves following actions:



Code Snippet 12 illustrates token generation and issuing in a .NET Core application using the `System.IdentityModel.Tokens.Jwt` library.

### Code Snippet 12

```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using Microsoft.IdentityModel.Tokens;
public class TokenService
{
    private readonly string _secretKey;
    private readonly string _issuer;
    public TokenService(string secretKey, string issuer)
    { _secretKey = secretKey;
        _issuer = issuer;
    }
```

```

public string GenerateToken(string userId, string username,
string[] roles) {
// Define claims
var claims = new[]
{ new Claim(JwtRegisteredClaimNames.Sub, userId),
new Claim(ClaimTypes.Name, username),
new Claim(ClaimTypes.Role, string.Join(", ", roles)),
new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString())
};
// Create credentials using the secret key
var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_secretKey));
var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);
// Create a JWT token
var token = new JwtSecurityToken(
 issuer,
 issuer,
 claims,
 expires: DateTime.Now.AddMinutes(30), // Token expiration
time
signingCredentials: creds );
// Serialize the token to a string
var tokenHandler = new JwtSecurityTokenHandler();
return tokenHandler.WriteToken(token);
}
}

...
// Example usage:
var tokenService = new TokenService("your_secret_key",
"your_issuer");
var token = tokenService.GenerateToken("123456", "john_doe",
new[] { "admin", "user" });
Console.WriteLine("Generated Token: " + token);

```

In this example:

- The `TokenService` class has a method `GenerateToken` that takes user information and generates a JWT.
- Claims include the subject (`Sub`), `username`, user roles, and a unique identifier (`Jti`).
- The token is signed using a symmetric key.
- The token has an expiration time (30 minutes in this example).
- The example usage demonstrates how to create an instance of `TokenService` and generate a token for a user with ID '123456,' `username` 'john\_doe', and roles 'admin' and 'user.'

## 12.7 Integrating Tokens from Identity Providers

When integrating authentication into .NET Core applications, leveraging tokens obtained from identity providers is a common and effective approach. Identity providers, such as OAuth 2.0 or OpenID Connect providers, authenticate users, and issue tokens that applications can use to authorize and identify users. Let us look at how to use tokens obtained from identity providers in .NET Core applications.

### Choose an Identity Provider as follows:

- OAuth 2.0 or OpenID Connect: Select an identity provider that supports OAuth 2.0 or OpenID Connect. Popular providers include Google, Facebook, Azure AD, and Okta.
- Obtain Client ID and Secret: Register the application with the chosen identity provider to obtain a client ID and secret.

### Then, configure Authentication in `Program.cs` using following steps:

- Install Necessary Packages: Use NuGet packages to install authentication-related libraries. For example, for OpenID Connect, one might use `Microsoft.AspNetCore.Authentication.OpenIdConnect`.
- In Visual Studio 2022, locate the `Program.cs` or equivalent file and configure services for authentication. The code in Code Snippet 13 is based on OpenID Connect.

#### Code Snippet 13

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = "https://your-identity-
provider.com";
    options.ClientId = "your-client-id";
    options.ClientSecret = "your-client-secret";
    options.CallbackPath = "/signin-oidc";
    // Additional configuration options...
});
```

The next action will be to authenticate users as follows:

- Redirect to Identity Provider: When a user accesses a protected resource, redirect them to the identity provider for authentication.

- Handle Callbacks: Configure callback routes where the identity provider sends the authentication result. Handle the callback to retrieve the token as shown in Code Snippet 14.

### Code Snippet 14

```
[HttpGet("/signin-oidc")]
public IActionResult SignInCallback()
{
    // Handle the callback to obtain user information and tokens
    return View("SignInCallback");
}
```

### Access Tokens and Identity Information using these steps:

- Access Tokens: Retrieve the access token from the authentication result. This token can be used to make authenticated requests to APIs as shown in Code Snippet 15.

### Code Snippet 15

```
var accessToken = await
    HttpContext.GetTokenAsync("access_token");
```

- Identity Information: Use the identity information to identify the user and obtain claims. Refer to Code Snippet 16.

### Code Snippet 16

```
var user = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
```

- When making requests to protected APIs, include the access token in the Authorization header. Refer to Code Snippet 17.

### Code Snippet 17

```
httpClient.DefaultRequestHeaders.Authorization = new
    AuthenticationHeaderValue("Bearer", accessToken);
```

Finally, logout from Identity Provider. When logging out of the application, also log the user out from the identity provider to end the user's session.

### Test and Debug the application.

Use Token Debuggers. During development, leverage token debugging tools provided by identity providers to inspect tokens and troubleshoot authentication issues.

## 12.8 Token-Based Authentication in MVC/Web Applications

Token-based authentication has become a prevalent method for securing MVC and Web applications, providing a stateless and scalable approach to user authentication. This method involves the generation and validation of tokens, allowing users to access protected resources.

### Understanding Token-Based Authentication

Statelessness is a distinguishing feature of token-based authentication compared to traditional session-based authentication. In this approach, each request includes a token, eliminating the necessity for server-side sessions. The token serves as proof of authentication, containing crucial information such as user ID, roles, and expiration time. It acts as an evidence that the user has been authenticated, facilitating secure and streamlined communication between the client.

### Token Components

Tokens, often JWTs, comprise three primary components: a header that specifies the token type and signing algorithm, a payload containing user claims, and a signature ensuring the integrity of the token. Claims, housed within the payload, offer information about the user, including their identity, roles, and additional attributes. These claims contribute to the comprehensive representation of the user within the token structure.

### Token Generation in MVC/Web Applications

Upon a user successfully logging in, an authentication success event is triggered, leading to the generation of a token. Ensure the token includes relevant claims, incorporating necessary user information for comprehensive authentication. Enhance security by signing the token using either a secret key or a private key, validating its authenticity and integrity.

### Token Issuing and Storage

Once generated, tokens are issued to the client, which can be a browser or a mobile app, and stored securely. Clients usually store tokens in cookies, local storage, or session storage.

### Token Validation in MVC/Web Applications

In MVC applications, token validation is often managed by middleware, which verifies the token's signature, expiration, and other claims. Additionally, authorization filters can be utilized to ensure that only authenticated users possessing valid tokens can access protected resources.

### Authorization and Access Control

Leverage claims within the token, such as user roles, to implement role-based access control. In MVC controllers, employ the [Authorize] attribute to restrict access to specific actions or controllers based on token authentication.

### **Logging Out and Token Revocation**

Implement mechanisms to handle token revocation in case a user logs out or their account is deactivated. This ensures that invalidated tokens cannot be used.

### **Benefits of Token-Based Authentication**

The stateless nature allows easy scalability, as there is no reliance on server-side sessions. Tokens facilitate cross-domain authentication, enabling users to access resources across different applications or services.

### **Secure Transmission**

To ensure the security of tokens during transmission, it is crucial to use HTTPS to encrypt data between the client and server.

# Summary

- ✓ Token-based authentication provides an additional encrypted security process in the authentication process.
- ✓ Token-based authentication is pivotal for securing and transmitting information in Web development.
- ✓ JWT consists of three parts: Header, Payload, and Signature.
- ✓ The Microsoft.AspNetCore.Authentication.JwtBearer package is utilized for JWT authentication.
- ✓ Token expiry is crucial, ensuring tokens are valid for a limited time.
- ✓ Secure transmission, typically over HTTPS, prevents interception and tampering.
- ✓ Protection of the secret key is essential to prevent unauthorized token creation.
- ✓ Choose an identity provider supporting OAuth 2.0 or OpenID Connect.
- ✓ Token-based authentication is secured and difficult to be hacked. This kind of authentication is used with APIs in mobile applications and SPAs.
- ✓ Tokens must be validated after it has been created. However, tokens can be validated only as a string.

# Test Your Knowledge



1. Token-based authentication is used with \_\_\_\_\_ in mobile applications and SPAs.

<b>A</b>	Classes
<b>B</b>	Objects
<b>C</b>	Attributes
<b>D</b>	APIs

2. Which of the following is NOT a component of a JWT?

<b>A</b>	Header
<b>B</b>	Payload
<b>C</b>	Footer
<b>D</b>	Number

3. In .NET Core Token Authentication, what is the purpose of Microsoft.AspNetCore.Authentication.JwtBearer package?

<b>A</b>	Token generation
<b>B</b>	Token validation
<b>C</b>	Token issuance
<b>D</b>	Token storage

4. What is a crucial security consideration when using token-based authentication?

<b>A</b>	Token generation
<b>B</b>	Token expiration
<b>C</b>	Token issuance
<b>D</b>	Token transmission

5. What is the primary purpose of the Header component in a JWT?

<b>A</b>	Carrying user claims
<b>B</b>	Ensuring token integrity
<b>C</b>	Containing metadata about the token and signing algorithm
<b>D</b>	Storing the token's expiration time

## Answers

1	D
2	D
3	B
4	B
5	C

## Try It Yourself

1. Create a basic ASP.NET Core Web application that implements token-based authentication. The task involves setting up the application, configuring token authentication, and securing endpoints.

### Hints:

- I. **For the User Model:**
  - a. Define a User class in the project with properties such as UserId, Username, and Password.
  - b. For simplicity, manually add a few user details directly in the class.
- II. **For Token Generation:**
  - a. Implement a TokenService class responsible for generating JWTs.
  - b. Utilize the System.IdentityModel.Tokens.Jwt library.
  - c. Include claims such as UserId and Username in the token payload.
  - d. Use a symmetric key for signing.
- III. **Authentication Middleware:**
  - a. In the Program.cs file, configure JWT authentication middleware.
  - b. Install the necessary NuGet packages, including Microsoft.AspNetCore.Authentication.JwtBearer.
  - c. Configure JWT authentication services.
  - d. Configure the authentication middleware.
- IV. **Authentication Logic:**
  - a. Create an endpoint (for example, /api/authenticate) that accepts user credentials (username and password) through a POST request.
  - b. Validate the received credentials against the hardcoded user details.
  - c. If the credentials are valid, generate a JWT using the TokenService and return it in the response.
- V. **Token Validation:**
  - a. Implement a middleware or filter to validate incoming JWTs for protected endpoints.
  - b. Ensure the token's signature and claims are verified.
  - c. Secure an endpoint (for example, /api/secure) using the [Authorize] attribute.
- VI. **Testing the Application:**
  - a. Use a tool such as Postman or curl to test the authentication.
  - b. Make requests to the authentication endpoint (/api/authenticate) with valid and invalid credentials.
  - c. Obtain the generated token and use it to access the secured endpoint (/api/secure).

# *Session 13: Deployment and Unit of Work Patterns in ASP.NET Core*

## **Session Overview**

This session describes repository pattern and unit of work. It also details the importance of using repository pattern and benefits of unit of work. It explains the difference between repository pattern and repository pattern with unit of work.

## **Objectives**

In this session, students will learn to:

- ✓ Describe repository pattern
- ✓ List advantages of repository pattern
- ✓ Explain the difference between repository pattern and repository pattern with Unit of Work
- ✓ List benefits of Unit of Work

## **13.1 Introduction**

In order to provide database connectivity using SQL Server or Oracle databases, following layers are used in traditional ASP.NET MVC 5.x Web applications:

- Model
- View
- Controller

Here, View is used for presentation or UI. The model is responsible for the database interaction and the controller for handling all the CRUD operations.

The model contains the ADO.NET EF which is an ORM. It is a part of the .NET framework. It creates business objects and entities based on database tables. It enables basic CRUD activities as well as the ability to manage relationships between entities including inheritance relationships.

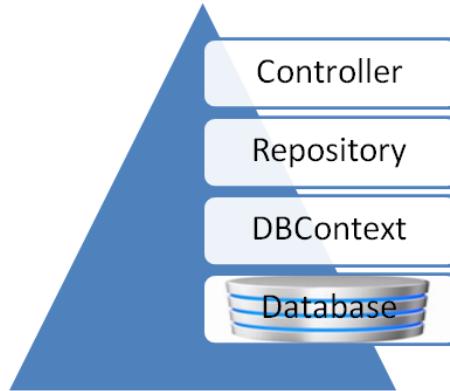
In ASP.NET Core applications, however, using **repository patterns** is recommended.

A **repository pattern** is a pattern that allows creation of an intermediary layer between data access layer and business logic layer. The repository pattern is also known as a data access pattern. Through a loosely linked approach, it enables access to data. The data access logic is plotted in a different class or series of classes called a repository, which oversees preserving the application's business model.

## **13.2 Repository Pattern**

A repository design allows full data access logic in a separate class called a repository. It is not on the controller. Therefore, a repository is a class that is defined for an entity that allows all CRUD activities to be performed on that entity.

For example, a repository for an entity *Department* will have standard CRUD activities as well as any other extra operations relevant to it. As shown in Figure 13.1, in a repository pattern, the controller is connected to `DbContext` via repository. `DbContext` is directly associated with the database.



**Figure 13.1: Repository Pattern**

Separating data and business layers provides many benefits. It ensures flexibility of the architecture. The logic is centralized and therefore, allows easy testing. Repository pattern can be of two types, namely, non-generic and generic.

### 13.2.1 Non-Generic Repository Pattern

All database actions relating to a specific entity are defined using non-generic repository approach within a separate class. For example, if there are two entities, Student and Employee, then two repositories for each entity, one for each entity's basic CRUD activities and one for each entity's additional operations, are established.

Consider a scenario where there is an entity called `Movies`. A repository is to be created for the entity `Movies`. Therefore, `DbContext` is used along with other CRUD operations as shown in Code Snippet 1. An ASP.NET Core Web app is assumed to have been created already with `Movies` and `MovieDbContext` classes.

#### Code Snippet 1

```
public class NonGenericRepository {
    public readonly MovieDbContext dbContext;
    private DbSet<Movies> entity;
    public NonGenericRepository () {
        dbContext = new MovieDbContext();
        entity = dbContext.Set<Movies>();
    }
    public void Insert(Movies entity) {
        entity.Add(entity);
        dbContext.SaveChanges();
    }
}
```

```

        public IEnumerable<Movies> GetAll() {
            return dbContext.Movies.ToList();
        }
        public void SaveChanges() {
            dbContext.SaveChanges();
        }
    }
}

```

In Code Snippet 1, a repository class named `NonGenericRepository` is created for `Movies`. In the code, an object of `DbContext` named `MoviesDbContext` is created. A `DBSet` object named `entity` is also declared. Within the constructor of class `NonGenericRepository`, the memory for the context is initialized. Memory for entity object is also set up. `Insert` method takes `Movies` as an argument and passes the same object to `Add` method of entities. `GetAll()` method returns the `IEnumerable` object that is fetched from `Movies.ToList()`.

`SaveChanges()` makes changes to the `DbContext` object.

### 13.2.2 Generic Repository Pattern

The generic repository pattern is used to provide common database activities, such as CRUD operations, for all database entities in a single class. A generic repository queries the data and then, maps the data to the business entity. Code Snippet 2 shows an example of a generic repository.

#### Code Snippet 2

```

public class GenericRepository<T> : IGenericRepository<T> where
T :
    BaseEntity {
    public readonly ApplicationDbContext dbContext;
    private DbSet<T> entities;
    public GenericRepository(ApplicationDbContext _dbcontext) {
        dbContext = _dbcontext;
        this.entities = dbContext.Set<T>();
    }
    public void Insert(T entity) {
        if (entity == null)
        {
            throw new ArgumentNullException("Entity
                Missing");
        }
        else
        {
            entities.Add(entity);
            dbContext.SaveChanges();
        }
    }
}

```

```

public IEnumerable<T> GetAll()
{
    return entities.AsEnumerable();
}
public void SaveChanges()
{
    dbContext.SaveChanges();
}
}

```

A GenericRepository repository class is created. This class is being implemented from `IGenericRepository`. Here, T identifier is used which is meant for referring to any type of class. T is replaced at runtime during execution. T is replaced automatically the moment the reference of the class is taken. The code for calling the `GenericRepository` class is as follows:

```

GenericRepository MoviesObject=new GenericRepository<Movies>();
GenericRepository TheatreObject=new GenericRepository<Theatre>();

```

In Code Snippet 2, an object of DB context called `ApplicationDbContext` is created which is defined at the application level. `DBSet` object is created for T type identifier. Within the constructor of class `GenericRepository`, the memory for the `ApplicationDbContext` object named `context` is initialized. Memory for entity object of T identifier is set up. `Insert` method is taking T type identifier as an argument and passing the same object to `Add` method of `entities` after validating whether something is present in the entity object of T Type. `GetAll()` method returns the `IEnumerable` object of T type identifier that is fetched from that specific entity in the form of a list. `SaveChanges()` is making changes to `dbContext` object applicable to the entity that is set in the constructor.

## Benefits of Repository Pattern

A repository works by sending queries to the data and mapping results. This is done through Data mapper. Through repositories, dependencies are removed and calling clients do not have to depend on the technology through which data is retrieved.

This allows flexibility in applications. Some of the benefits are as follows:

Easier Testing	Concerns are separated	Loose connection
With separation of layers, testing is easier.	Separate application functionalities depending on function makes the code easier to evolve and maintain.	When switching databases, changes are made only in the data layer but not in the data access layer. The number of changes or adjustments required in code specifically in the data access layer are reduced.

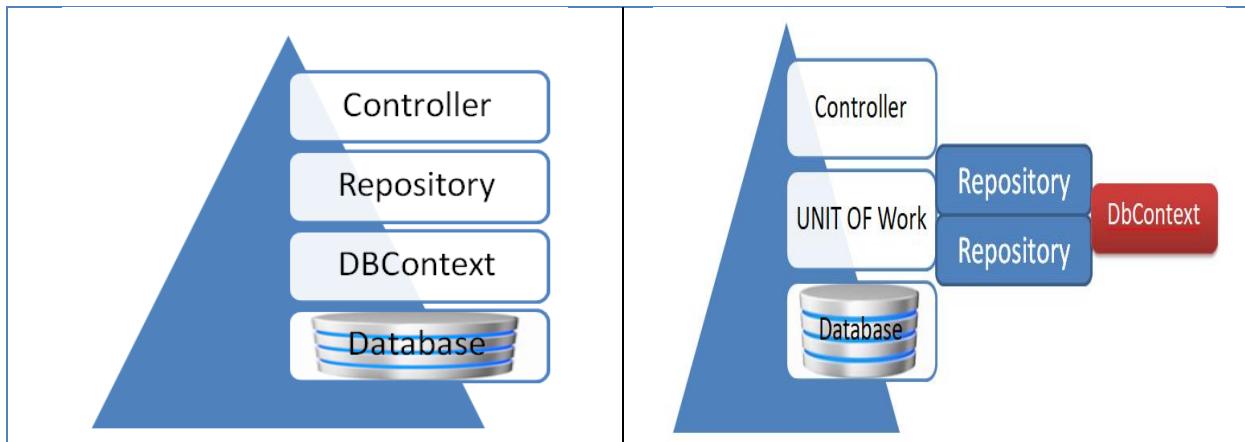
However, there are certain points that must be considered while implementing repositories. Sometimes, the code is difficult to understand because the number of classes are increased in spite of no redundant code. It may also raise performance related issues in some heavy systems.

### 13.3 Unit of Work

The concept of a Unit of Work (UoW) is important for successful implementation of a repository pattern. The UoW is a type of business transaction that combines all CRUD transactions, such as insert/update/delete into one. This ensures that the transactions are completed once instead of multiple transactions.

For all the changes, only one commit is made. Any transaction that does not guarantee data integrity is rolled back. UoW does not block any data table till all changes are committed.

Figure 13.2 shows the difference between repository pattern and repository pattern with UoW.



**Figure 13.2: Repository and Repository with UoW**

In Figure 13.2, the controller is concise because it contains one repository that via `DbContext` is connected to the database. When there are multiple repositories, each repository will have a corresponding entity. In this case, each repository will have and maintain its own instance of the `DbContext`. Sometimes, `SaveChanges()` of a single repository's fails while the other succeeds. This leads to database inconsistency.

To address the issue of data inconsistency, a layer between the controller and the repository is added. It receives the `DbContext` instance. The `DbContext` class holds the `DbSet` entities and is referred to as a UoW pattern. The `DbContext` class manages the database operations on these entities. It saves them as a single transaction to the database. This ensures that a transaction that runs for multiple repositories either

fails or succeeds completely. UoW ensures complete separation of the service layer from the data layer.

Code Snippet 3 shows an example of UoW for the same scenario of ASP.NET Core Web app with Movies entity.

### Code Snippet 3

```
public interface IUnitOfWork : IDisposable
{
    IMoviesRepository Movies { get; }
    ITheatreRepository Theatres { get; }

    int SaveChanges();
}

public class UnitOfWork : IUnitOfWork
{
    private readonly DatabaseContext db;

    public UnitOfWork()
    {
        db = new DatabaseContext();
    }

    private IMoviesRepository _Movies;
    public IMoviesRepository Movies
    {
        get
        {
            if (this._Movies == null)
            {
                this._Movies = new MoviesRepository(db);
            }
            return this._Movies;
        }
    }

    private ITheatreRepository _Theatres;
    public ITheatreRepository Theatres
    {
        get
        {
            if (this._Theatres == null)
            {
                this._Theatres = new TheatreRepository(db);
            }
            return this._Theatres;
        }
    }
}
```

```
public int SaveChanges()
{
    return db.SaveChanges();
}

public void Dispose()
{
    db.Dispose();
}
```

In Code Snippet 3, there are two repositories, `MoviesRepository` and `TheatreRepository`. A class named `UnitOfWork` is used to manage the two repositories.

### 13.3.1 Benefits of Unit of Work

Implementing repository pattern with UoW has various benefits. Some of the benefits are as follows:

- Ensures flexibility in architecture because business and data layers are separate.
- Ensures ease of testability through implementation of unit testing.
- Increased abstraction due to separation of business logic and database.
- No redundancy in code though number of classes are more.
- Easy management of in-memory database operations through a single transaction.

# Summary

- ✓ The repository pattern serves as an intermediary layer between an application's data access layer and its business logic layer.
- ✓ All database actions relating to a specific entity are defined using the non-generic repository approach within a separate class.
- ✓ The generic repository pattern is used to provide common database activities, such as CRUD operations, for all database entities in a single class.
- ✓ The UoW combines all CRUD transactions, such as insert/update/delete into one.
- ✓ UoW does not block any data table till it commits the changes.

# Test Your Knowledge



1. The repository pattern serves as an intermediary layer between Controller layer and which of these layers?
4. What combines all CRUD transactions?

<b>A</b>	DbContext
<b>B</b>	Controller
<b>C</b>	Database
<b>C</b>	None of these

2. Which approach is used to define database actions relating to a specific entity within a separate class?

<b>A</b>	Non-Generic Repository
<b>B</b>	Generic Repository
<b>C</b>	MVC
<b>D</b>	MVVM

3. Which approach is used to provide common database activities?

<b>A</b>	Non-Generic Repository
<b>B</b>	Generic Repository

5. The database states are managed by the UoW pattern.

<b>A</b>	True
<b>B</b>	False

## Answers

1	A
2	A
3	B
4	C
5	A

## **Try It Yourself**

1. Create a non-generic repository for a hypothetical entity 'Product' in an ASP.NET Core application. Include methods for Insert, GetAll, and SaveChanges. Assume the existence of an Entity Framework DbContext named ProductDbContext.
2. Implement a generic repository that can handle CRUD operations for any entity type. Use the generic repository to manage a hypothetical entity 'Customer' in an ASP.NET Core application.

# *Session 14: User Login and ASP.NET Core Identity*

## **Session Overview**

This session outlines fundamental concepts of user identity management and authentication in ASP.NET Core applications

## **Objectives**

In this session, students will learn to:

- ✓ Describe User Identity and Login
- ✓ Explain Password Security
- ✓ Describe Session State in ASP.NET Core
- ✓ Elaborate on User Identity and Security with example application

## **14.1 Introduction to User Identity and Login**

User identity refers to the information associated with an individual user within a system or application. It encompasses various attributes and characteristics that uniquely identify a user and determine their access rights, permissions, and interactions within the system. User identity plays a crucial role in ensuring security, personalization, and accountability in software applications and online services.

### **Importance of User Identity**

1. **Access Control:** User identity is fundamental to access control mechanisms within an application or system. It helps in enforcing authentication and authorization policies to determine what resources or functionalities a user can access based on their identity and associated permissions.
2. **Security:** Proper management of user identity is essential for maintaining the security of sensitive data and resources. By accurately identifying users and authenticating their identities, organizations can mitigate the risk of unauthorized access and potential security breaches.
3. **Personalization:** User identity enables applications to personalize user experiences by tailoring content, settings, and recommendations based on individual preferences, past behavior, and demographic information associated with their identity.
4. **Auditing and Accountability:** User identity allows organizations to track and audit user actions within the system, facilitating accountability and compliance with regulatory requirements. By associating user activities with specific identities, organizations can trace back any unauthorized or malicious actions to the responsible users.

**5. Single Sign-On (SSO) and Federated Identity:** User identity management systems enable features such as Single Sign-On (SSO), which streamline the authentication process across multiple applications or services. Federated identity solutions extend this capability to allow users to use their credentials from one identity provider to access resources from other trusted service providers.

### Different Types of User Identities

#### Local User Identity:

Local user identities are specific to a single application or system and are managed internally by that application. Users typically register directly with the application, providing credentials such as a username and password to authenticate their identity.

#### External Identity Providers:

External identity providers, such as social media platforms (for example, Google, Facebook, and Twitter) or enterprise identity providers (for example, Active Directory or LDAP), offer identity management services that can be integrated with applications. Users authenticate with these external providers, and the applications trust the authentication tokens provided by them, simplifying the login process and leveraging existing user identities.

#### Service Accounts:

Service accounts represent non-human users, such as system processes, scripts, or automated services, which require access to resources within an application or system. These accounts are typically used for machine-to-machine communication and are authenticated using API keys, OAuth tokens, or other authentication mechanisms.

#### Anonymous Identities:

Anonymous identities are temporary or pseudonymous identities assigned to users who have not authenticated themselves. They are commonly used for tracking user sessions or providing limited access to resources without requiring full authentication.

#### Role-Based Identities:

Role-based identities are identities associated with specific roles or groups within an organization or application. Users are granted access permissions based on their roles, and their identities may include additional attributes or metadata related to their roles.

## 14.2 Understanding User Login

Authentication is the process of verifying the identity of a user attempting to access a system or application.

#### **14.2.1 Authentication Process**

In ASP.NET Core, authentication typically involves following steps:

- 1. User Request:**
  - The user sends a request to access a protected resource or perform a privileged action within the application.
- 2. Authentication Middleware:**
  - The request is intercepted by authentication middleware configured in the ASP.NET Core application pipeline.
  - The middleware examines the request for authentication-related information, such as tokens or credentials.
- 3. Identity Verification:**
  - The authentication middleware verifies the user's identity based on the provided credentials or tokens.
  - This verification process may involve comparing the credentials against stored user data, validating tokens with an identity provider, or using other authentication mechanisms.
- 4. Authentication Result:**
  - Upon successful verification, the user's identity is established, and an authentication ticket or principal object is created.
  - This object typically contains information about the authenticated user, such as their claims, roles, or identity properties.
- 5. Access Authorization:**
  - Once authenticated, the request is passed down the ASP.NET Core pipeline for authorization.
  - Authorization middleware evaluates the user's identity and associated permissions to determine whether they are authorized to access the requested resource or perform the action.

#### **14.2.2 Importance of Secure Login Mechanisms**

Secure login mechanisms are essential for protecting user accounts, sensitive data, and the overall integrity of an application. Here is why they are crucial:

- 1. Preventing Unauthorized Access:**
  - Secure login mechanisms help ensure that only authorized users can access protected resources or perform privileged actions within the application.
  - By verifying user identities and enforcing access controls, these mechanisms mitigate the risk of unauthorized access by malicious actors.
- 2. Protecting User Credentials:**
  - Secure login mechanisms employ encryption, hashing, and other cryptographic techniques to safeguard user credentials, such as passwords or authentication tokens, during transmission and storage.
  - This helps prevent unauthorized interception or disclosure of sensitive information that could compromise user accounts.

### **3. Mitigating Credential-Based Attacks:**

- Strong authentication mechanisms, such as Multi-Factor Authentication (MFA) or CAPTCHA challenges, can help mitigate common credential-based attacks, such as brute force attacks, credential stuffing, or phishing.
- These mechanisms add additional layers of security beyond traditional username/password authentication, making it harder for attackers to compromise user accounts.

### **4. Building User Trust:**

- Implementing robust security measures instills confidence and trust in users, assuring them that their accounts and data are protected from unauthorized access and misuse.
- This positive user experience fosters loyalty and encourages continued engagement with the application or service.

## **14.3 Password Security**

Now, let us explore password security best practices and implementing them in ASP.NET Core, focusing on using ASP.NET Core Identity for secure password management.

### **Password Hashing**

Password hashing is a critical aspect of password security in Web applications. It involves converting a plain-text password into a hashed value using a cryptographic hashing algorithm. By hashing passwords, we ensure that they are not stored in plain text in the database, significantly enhancing security.

Commonly used hashing algorithms include SHA-256, bcrypt, and Argon2. It is crucial to employ a strong, adaptive hashing algorithm specifically designed for password storage to resist brute-force attacks effectively.

### **Salted Hashing**

Salting is a process in securing content. It is like adding extra security layers to protect passwords. Imagine each password as a special recipe and salting is similar to adding a secret ingredient to make it unique. This secret ingredient, called a salt, is just a random value. So, before we turn the password into a scrambled mess (hash), we mix it with this salt.

Why do we do this?

Imagine if everyone's password was just hashed by itself. It would be easier for bad guys to guess common passwords because they could use pre-made lists of hashed passwords. However with salting, even if two people have the same password, their hashed versions will look completely different because of the unique salt added to each one.

So, salting is similar to adding a secret twist to each password, making them more secure against hackers who try to guess passwords using fancy techniques. Remember that we store both the salt and the hashed password in the database so we can check passwords properly when someone tries to log in.

Salted hashing adds an additional layer of security to password storage. It involves appending a random value called a salt to each password before hashing it. Salting prevents attackers from efficiently cracking passwords using precomputed hash tables, commonly known as rainbow tables. Each user's salt should be unique, and both the salt and hashed password should be stored in the database to facilitate password verification during login.

### **Password Policies and Complexity Requirements**

In addition to hashing and salting, enforcing password policies and complexity requirements is essential for bolstering security. Password policies establish rules for password complexity, length, and other attributes.

Common requirements include specifying a minimum password length, mandating a mix of uppercase and lowercase letters, digits, and special characters. Implementing password expiration policies and enforcing users to change passwords periodically can further enhance security by reducing the likelihood of compromised passwords being exploited over time. These measures collectively contribute to a robust password security framework, mitigating the risk of unauthorized access and data breaches in Web applications.

## **14.4 Session State in ASP.NET Core**

Session state management is crucial for Web applications to maintain user-specific data across multiple requests. It allows applications to store user-related information temporarily, such as user authentication tokens, shopping cart items, user preferences, and so on. Session state ensures a personalized and seamless user experience by preserving user interactions during their visit to the application.

Key reasons for the importance of session state include:

1. **User Personalization:** Session state enables personalized experiences by storing user-specific data such as shopping cart contents, user preferences, and browsing history.
2. **Data Persistence:** It allows Web applications to retain data temporarily without storing it permanently in a database, reducing the requirement for frequent database accesses and enhancing performance.
3. **Authentication and Authorization:** Session state often plays a role in managing user authentication tokens or authorization claims, allowing applications to maintain user identity across requests.

Different approaches to session management cater to various application requirements and scalability requirements.

**In-Memory Session Storage** involves storing session data directly within the application's memory. This approach is straightforward and suitable for small-scale applications with a single server instance. However, it is not recommended for large-scale or distributed applications due to scalability limitations and potential data loss on server restarts. In-memory session storage may lead to performance issues and is not resilient to server failures.

ASP.NET Core provides built-in support for in-memory session state using the `MemoryCache` or `IDistributedCache` interfaces.

**Distributed Session Storage**, on the other hand, distributes session data across multiple servers or external data stores, providing scalability and resilience. Common options for distributed session storage include databases, distributed cache systems such as Redis, or cloud-based storage services such as Azure Cosmos DB. By spreading session data across multiple nodes, distributed session storage ensures availability even in scenarios involving server failures or restarts. This approach facilitates seamless scalability and improved fault tolerance, making it suitable for applications with high availability requirements and fluctuating workloads.

#### 14.4.1 Implementing Session State in ASP.NET Core

ASP.NET Core provides middleware components for managing session state within the request processing pipeline. To enable session state, you are required to configure and use the appropriate middleware in your application and choose a storage mechanism for the session data.

Example of enabling session state middleware in `Program.cs` is shown in Code Snippet 1.

##### Code Snippet 1

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using System;
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.ConfigureServices(services =>
                {
                    services.AddControllersWithViews();
                });
            });
}
```

```

        // Configure session options
        services.AddSession(options =>
        {
            options.Cookie.HttpOnly = true;
            options.Cookie.IsEssential = true;
            options.IdleTimeout =
                TimeSpan.FromMinutes(30); // Adjust timeout as required
        });
    .Configure(app =>
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
        // Use session middleware
        app.UseSession();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern:
                    "{controller=Home}/{action=Index}/{id?}");
        });
    });
}

```

In this implementation:

- `AddSession` method is used to configure session options such as cookie properties and timeout period.
- `UseSession` middleware is added to the request pipeline to enable session state.
- You can adjust session options such as timeout period according to your application's requirements.

Let us break down the provided implementation:

### 1. `AddSession` Method:

- `services.AddSession(options => { ... })` is used to configure session options.
- Inside the lambda expression, you can specify various settings for session management.
- In this example:
  - `options.Cookie.HttpOnly = true;`: Sets the `HttpOnly` property of the session cookie to true, preventing client-side JavaScript from accessing the cookie.
  - `options.Cookie.IsEssential = true;`: Marks the session cookie as essential. This means that the session middleware will

treat requests that fail to send the session cookie as a failure, preventing the request from being processed further.

- `options.IdleTimeout = TimeSpan.FromMinutes(30);` : Sets the idle timeout period for the session. After the specified duration of inactivity, the session will expire and be removed.

## 2. UseSession Middleware:

- `app.UseSession()` is added to the request pipeline inside the `Configure` method.
- This middleware enables session state for the application.
- It should be placed after other middleware such as routing, static files, and authentication, but before MVC endpoints.
- This ensures that session data is available for handling requests in controllers and views.

## 3. Adjusting Session Options:

- The provided implementation allows you to adjust session options according to your application's requirements.
- For example, you can change the timeout period (`options.IdleTimeout`) to control how long sessions remain active during periods of inactivity.
- You can also configure other properties of the session cookie, such as domain, path, secure, and sameSite settings, as required .

## 14.5 User Identity and Security with Example Application

Let us walk through a step-by-step guide to implementing user identity and security features in an ASP.NET Core Web application using a complete example. This example will cover user registration, login, and registration.

### Step 1: Create a new ASP.NET Core Project

Create a new ASP.NET Core Model View Controller project as shown in Figure 14.1.

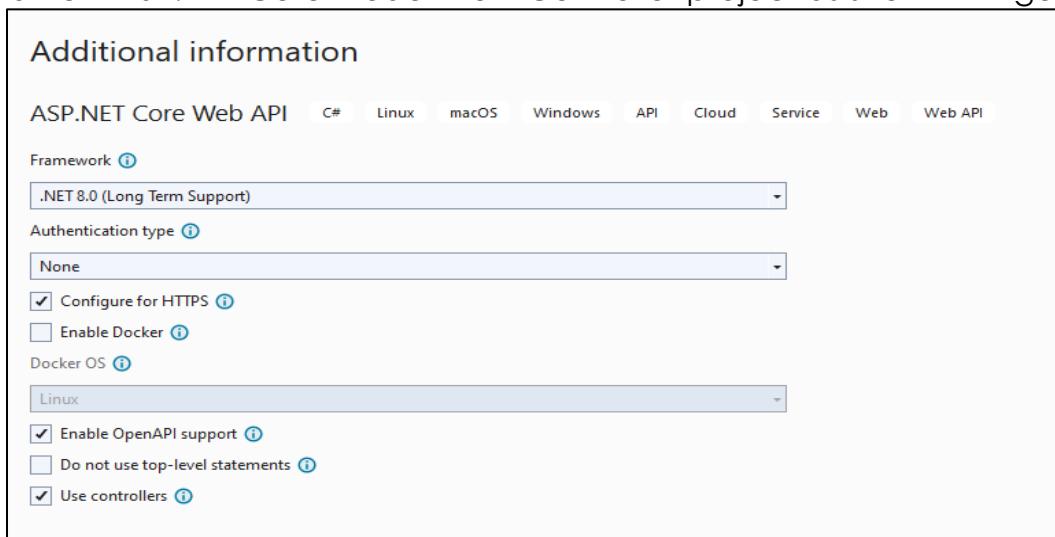


Figure 14.1: ASP.NET Core Project Setup

It is assumed here that the application is named UserIdentity7 but you can specify a different name based on your requirement.

## Step 2: Setting up ASP.NET Core Identity

First, let us set up ASP.NET Core Identity to handle user authentication and authorization.

Install the required packages using NuGet Package Manager in Visual Studio 2022:

```
Microsoft.AspNetCore.Identity.EntityFrameworkCore  
Microsoft.EntityFrameworkCore  
Microsoft.EntityFrameworkCore.SqlServer  
Microsoft.EntityFrameworkCore.Tools
```

## Step 3: Set up Controller Classes

Define an AccountController class in Controllers folder as shown in Code Snippet 2.

This code defines an AccountController responsible for handling user authentication actions such as login, registration, and logout in an ASP.NET Core MVC application. It utilizes ASP.NET Core Identity's SignInManager and UserManager to authenticate users and manage user accounts, enabling features such as password hashing, account creation, and session management.

### Code Snippet 2

```
using Microsoft.AspNetCore.Identity;  
using Microsoft.AspNetCore.Mvc;  
using System;  
using UserIdentity7.ViewModels;  
using UserIdentity7.Models;  
  
namespace UserIdentity7.Controllers  
{  
    public class AccountController(SignInManager<AppUser>  
signInManager, UserManager<AppUser> userManager) : Controller  
    {  
        public IActionResult Login(string? returnUrl = null)  
        {  
            ViewData["ReturnUrl"] = returnUrl;  
            return View();  
        }  
  
        [HttpPost]  
        public async Task<IActionResult> Login(LoginVM model,  
string? returnUrl = null)  
        {  
            ViewData["ReturnUrl"] = returnUrl;  
            if (ModelState.IsValid)
```

```
        {
            //login
            var result = await
signInManager.PasswordSignInAsync(model.Username!,
model.Password!, model.RememberMe, false);

            if (result.Succeeded)
            {
                return RedirectToAction(returnUrl);
            }

            ModelState.AddModelError("", "Invalid login attempt");
            }
            return View(model);
        }

public IActionResult Register(string? returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}

[HttpPost]
public async Task<IActionResult> Register(RegisterVM model,
string? returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        AppUser user = new()
        {
            Name = model.Name,
            UserName = model.Email,
            Email = model.Email,
            Address = model.Address
        };

        var result = await userManager.CreateAsync(user,
model.Password!);

        if (result.Succeeded)
        {
            await signInManager.SignInAsync(user, false);

            return RedirectToAction(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError("", error.Description);
        }
    }
}
```

```

        }
    }
    return View(model);
}

public async Task<IActionResult> Logout()
{
    await signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}

private IActionResult RedirectToLocal(string? returnUrl)
{
    return !string.IsNullOrEmpty(returnUrl) &&
Url.IsLocalUrl(returnUrl)
    ? Redirect(returnUrl)
    : RedirectToAction(nameof(HomeController.Index),
nameof(HomeController));
}
}
}
}

```

Define a `HomeController` class in **Controllers** folder as shown in Code Snippet 3. This code defines a `HomeController` responsible for rendering views related to the home page, privacy policy, and error handling in an ASP.NET Core MVC application. It is decorated with the `[Authorize]` attribute, ensuring that only authenticated users can access the actions within this controller, thereby enforcing access control measures.

### Code Snippet 3

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;
using UserIdentity7.Models;

namespace UserIdentity7.Controllers
{
    [Authorize]
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Privacy()
        {
            return View();
        }
    }
}

```

```
[ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

#### Step 4: Set up DbContext

Define a DbContext class to represent your application's database context. This class will inherit from `IdentityDbContext<IdentityUser>` as shown in Code Snippet 4.

#### Code Snippet 4

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace UserIdentity7.Data
{
    public class AppDbContext : IdentityDbContext<IdentityUser>
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
        {
        }
    }
}
```

This code defines a class named `AppDbContext` that serves as the database context for the ASP.NET Core Identity system and user management. It inherits from `IdentityDbContext<IdentityUser>`, indicating that it provides access to the tables and functionality required for storing and managing user accounts and related data in the database.

#### Step 5: Define ApplicationUser Class

Create a class named `AppUser` representing the application's user. This class will inherit from `IdentityUser`. Refer to Code Snippet 5. Add this code to the class.

#### Code Snippet 5

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

namespace UserIdentity7.Models;

public class AppUser : IdentityUser
```

```
{
    [StringLength(100)]
    [MaxLength(100)]
    [Required]
    public string? Name { get; set; }
    public string? Address { get; set; }
}
```

This code defines a custom user model named `AppUser` that inherits from `IdentityUser`, which is provided by ASP.NET Core Identity for managing user accounts. It adds additional properties such as `Name` and `Address` to the user model, allowing for the storage of extra user information beyond the default properties provided by `IdentityUser`.

### **Step 6: Define Login View Model and Register Model Class**

Create a class named `LoginVM` in `ViewModels` folder. Add to it the code given in Code Snippet 6. This code defines a view model class named `LoginVM` used for handling user login data in the ASP.NET Core application. It includes properties for `Username`, `Password`, and `RememberMe`, each adorned with data annotations such as `[Required]` and `[Display]` to specify validation rules and display attributes for the corresponding input fields in the login form.

#### **Code Snippet 6**

```
using System.ComponentModel.DataAnnotations;

namespace UserIdentity7.ViewModels
{
    public class LoginVM
    {
        [Required(ErrorMessage = "Username is required.")]
        public string? Username { get; set; }

        [Required(ErrorMessage = "Password is required.")]
        [DataType(DataType.Password)]
        public string? Password { get; set; }

        [Display(Name = "Remember Me")]
        public bool RememberMe { get; set; }
    }
}
```

Create a class named `RegisterVM` in `ViewModels` folder. Add to it the code given in Code Snippet 7. This code defines a view model class named `RegisterVM` used for handling user registration data in the ASP.NET Core application. It includes properties for `Name`, `Email`, `Password`, `ConfirmPassword`, and `Address` each adorned with data annotations such as `[Required]`, `[DataType]`, and `[Compare]`

to specify validation rules and display attributes for the corresponding input fields in the registration form.

### Code Snippet 7

```
using System.ComponentModel.DataAnnotations;

namespace UserIdentity7.ViewModels
{
    public class RegisterVM
    {
        [Required]
        public string? Name { get; set; }

        [Required]
        [DataType(DataType.EmailAddress)]
        public string? Email { get; set; }

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

        [Compare("Password", ErrorMessage = "Passwords don't
            match.")]
        [Display(Name = "Confirm Password")]
        [DataType(DataType.Password)]
        public string? ConfirmPassword { get; set; }

        [DataType(DataType.MultilineText)]
        public string? Address { get; set; }
    }
}
```

### Step 7: Create Login and Register cshtml Files

Create a new Razor view file named `Login.cshtml` inside the **Views/Account** folder. This view will contain a form for user login as shown in Code Snippet 8.

### Code Snippet 8

```
@using UserIdentity7.ViewModels;
@model LoginVM

 @{
    ViewData["Title"] = "Login";
}

<div class="row d-flex align-items-center justify-content-center">
    <div class="col-md-3 card p-3">
        <h2 class="text-center text-info">Login</h2>
        <form asp-action="Login" method="post" asp-route-
returnurl="@ViewData["ReturnUrl"]">
```

```

        <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
            <div class="mb-1">
                <label asp-for="Username" class="control-
label"></label>
                    <input asp-for="Username" class="form-control" />
                    <span asp-validation-for="Username" class="text-
danger"></span>
                </div>
                <div class="mb-1">
                    <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="mb-1 form-check">
                        <label class="form-check-label">
                            <input asp-for="RememberMe" class="form-check-
input" /> @Html.DisplayNameFor(a => a.RememberMe)
                        </label>
                    </div>
                    <div class="row">
                        <div class="col-8">
                            <a asp-action="Register" class="text-
decoration-none float-start mt-2" asp-route-
returnurl="@ ViewData["ReturnUrl"]">Don't have account?</a>
                        </div>
                        <div class="col-4">
                            <input type="submit" value="Login" class="btn
btn-primary btn-sm float-end" />
                        </div>
                    </div>
                </div>
            </form>
        </div>
    </div>

@section Scripts {
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}

```

This code defines a Razor view named `Login.cshtml` responsible for rendering the user login form in the ASP.NET Core MVC application. It utilizes the `LoginVM` view model to bind and validate user input fields such as `Username` and `Password` and includes logic to display validation errors and handle form submission using the `asp-action` attribute to specify the action method for form submission.

Create a new Razor view file named Register.cshtml inside the **Views/Account** folder. This view will contain a form for user register as shown in Code Snippet 9.

### Code Snippet 9

```
@using UserIdentity7.ViewModels;
@model RegisterVM

{@{
    ViewData["Title"] = "Register";
}

<div class="row d-flex align-items-center justify-content-center">
    <div class="col-md-4 card p-3">
        <h2 class="text-center text-info">Register</h2>
        <form asp-action="Register" method="post" asp-route-
returnurl="@ViewData["ReturnUrl"]">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
            <div class="mb-1">
                <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="mb-1">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-
danger"></span>
            </div>
            <div class="mb-1">
                <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="mb-1">
                <label asp-for="ConfirmPassword" class="control-
label"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-
danger"></span>
            </div>
            <div class="mb-1">
                <label asp-for="Address" class="control-
label"></label>
            </div>
        </form>
    </div>
</div>
```

```

        <textarea asp-for="Address" class="form-control"
rows="2"></textarea>
        <span asp-validation-for="Address" class="text-
danger"></span>
    </div>
    <div class="row">
        <div class="col-8">
            <a asp-action="Login" asp-route-
returnurl="@ViewData["ReturnUrl"]" class="text-decoration-none
float-start mt-2">Have account?</a>
        </div>
        <div class="col-4">
            <input type="submit" value="Register"
class="btn btn-primary btn-sm float-end" />
        </div>
    </div>
</form>
</div>
</div>
@section Scripts {
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}

```

This code defines a Razor view named `Register.cshtml` responsible for rendering the user registration form in the ASP.NET Core MVC application. It utilizes the `RegisterVM` view model to bind and validate user input fields such as `Name`, `Email`, `Password`, `ConfirmPassword`, and `Address`, and includes logic to display validation errors and handle form submission using the `asp-action` attribute to specify the action method for form submission.

Create a new Razor view file named `_LoginPartial.cshtml` inside the **Views/Shared** folder. This code defines a partial view responsible for rendering the navigation bar in the ASP.NET Core MVC application. It includes logic to display different navigation options based on whether the user is signed in or not. If the user is signed in, it displays a drop-down menu with the user's name and a logout option. If the user is not signed in, it displays options to register and login. Refer to Code Snippet 10.

#### Code Snippet 10

```

@using UserIdentity7.Controllers;
@using UserIdentity7.Models;
@using Microsoft.AspNetCore.Identity;
@inject SignInManager<AppUser> signInManager
<ul class="navbar-nav">
    @if (signInManager.IsSignedIn(User))
    {

```

```

<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">
        @User.Identity!.Name!
    </a>
    <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
        <li>
            <a class="dropdown-item" asp-action="Logout" asp-controller="Account"> Logout</a>
        </li>
    </ul>
</li>
}
else {
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Account" asp-action="Register">Register</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Account" asp-action="Login">Login</a>
</li>
}
</ul>

```

Include the partial name in your \_Layout.cshtml file which is present in Views/Shared as shown in Figure 14.2.

```

1
13 <header>
14     <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
15         <div class="container">
16             <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index" href="#">Sample Application</a>
17             <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
18                 <span class="navbar-toggler-icon"></span>
19             </button>
20             <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
21                 <ul class="navbar-nav flex-grow-1">
22                     <li class="nav-item">
23                         <a class="nav-link text-dark" asp-area="" asp-controller="Account" asp-action="Register">Register</a>
24                     </li>
25                     <li class="nav-item">
26                         <a class="nav-link text-dark" asp-area="" asp-controller="Account" asp-action="Login">Login</a>
27                     </li>
28                 </ul>
29                 <partial name='_LoginPartial' />
30             </div>
31         </div>
32     </nav>
33 </header>
34 <div class="container">
35     <main role="main" class="pb-3">
36         @RenderBody()
37     </main>

```

**Figure 14.2: \_Layout File**

In this code, `<partial name="_LoginPartial" />` is added to include the login partial view within the layout view. This allows the navigation bar to dynamically display login-related options based on the user's authentication status.

The purpose of adding this partial view is to keep the navigation bar clean and modular. By including the login partial view, the navigation bar can easily adapt to and display different options for authenticated and unauthenticated users without cluttering the main layout view. This promotes code reusability and maintainability by separating concerns and keeping the codebase organized.

### Step 8: Configure services in Program.cs:

Now, let us incorporate the service configurations in `Program.cs` as shown in Code Snippet 11.

#### Code Snippet 11

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using UserIdentity7.Data;
using UserIdentity7.Models;

var builder = WebApplication.CreateBuilder(args);
var connectionString =
builder.Configuration.GetConnectionString("default");
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<AppDbContext>(
    options => options.UseSqlServer(connectionString));

builder.Services.AddIdentity<AppUser, IdentityRole>(
    options =>
    {
        options.Password.RequiredUniqueChars = 0;
        options.Password.RequireUppercase = false;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireLowercase = false;
    })
    .AddEntityFrameworkStores<AppDbContext>().AddDefaultTokenProv
iders();

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();

```

This code configures services and middleware for an ASP.NET Core application using the builder pattern. It sets up dependencies such as controllers, DbContext, Identity, and Entity Framework for the application. It configures the HTTP request pipeline by adding middleware for HTTPS redirection, static files serving, routing, authorization, and error handling. Additionally, it defines a default controller route for handling incoming requests.

Make sure to replace the SQL Server name highlighted with your own local SQL Server name in `appsetting.json` file as shown in Figure 14.3.

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
      "default": "Data Source=[REDACTED];Initial Catalog=CustomIdentityDB;Integrated Security=True;TrustServerCertificate=True"
    }
  }
}

```

**Figure 14.3: appsetting.json File**

This JSON configuration file specifies logging settings for the application, setting the default log level to Information and suppressing warnings from `Microsoft.AspNetCore` namespace. It also defines the connection string named `default`, which is used to connect to the SQL Server database named `CustomIdentityDB` on the specified data source, using integrated security and trusting the server certificate.

Final Project Structure in Solution Explorer is shown in Figure 14.4.



**Figure 14.4: Solution Explorer of Project**

Make sure to perform migration by using `add-migration Initial` command from package manager console of your project as shown in Figure 14.5. The `add-migration Initial` command is used in Entity Framework Core to scaffold a new migration named **Initial**. This command generates code to create a new database schema based on changes made to the `DbContext`. It analyzes the current state of the `DbContext` and generates a migration file containing the necessary operations to update the database schema to match the current model state. This command helps in managing database schema changes and keeping the database in sync with the application's data model.

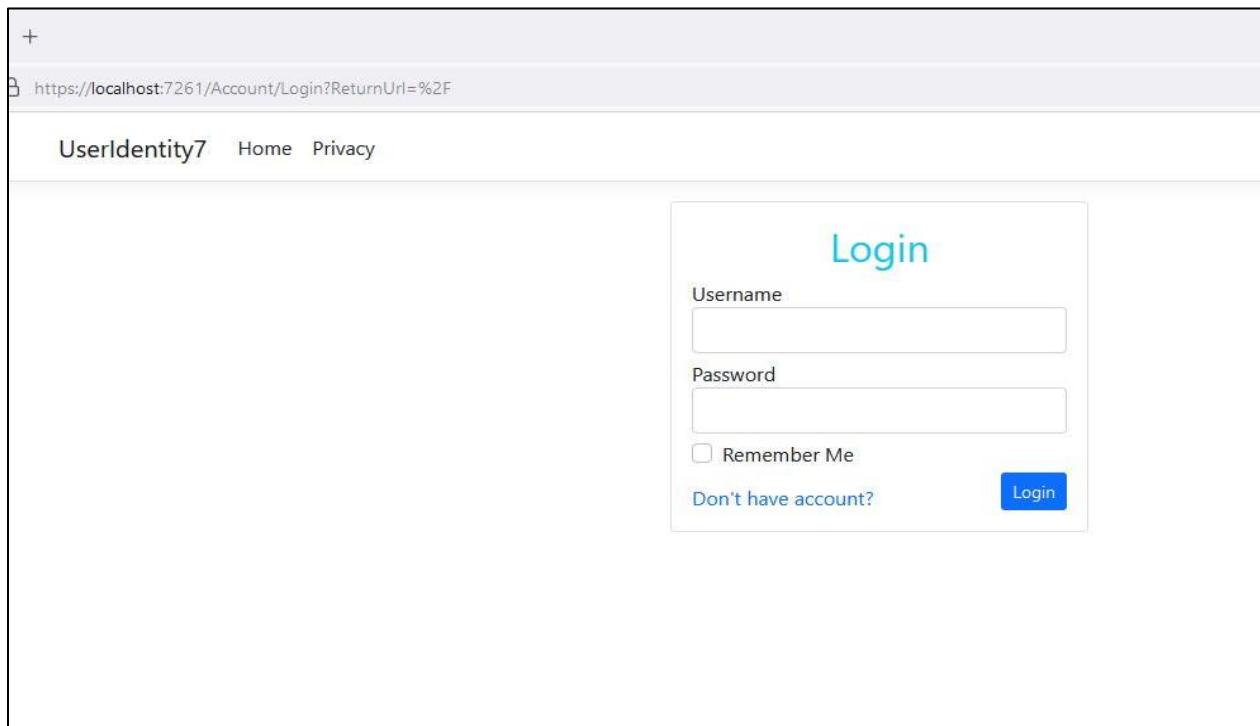
```
PM> add-migration Initial
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

**Figure 14.5: Migration Command**

Finally, run Update-Database command in the package manager console. This will autogenerate the necessary User identity tables required for ASP.NET.

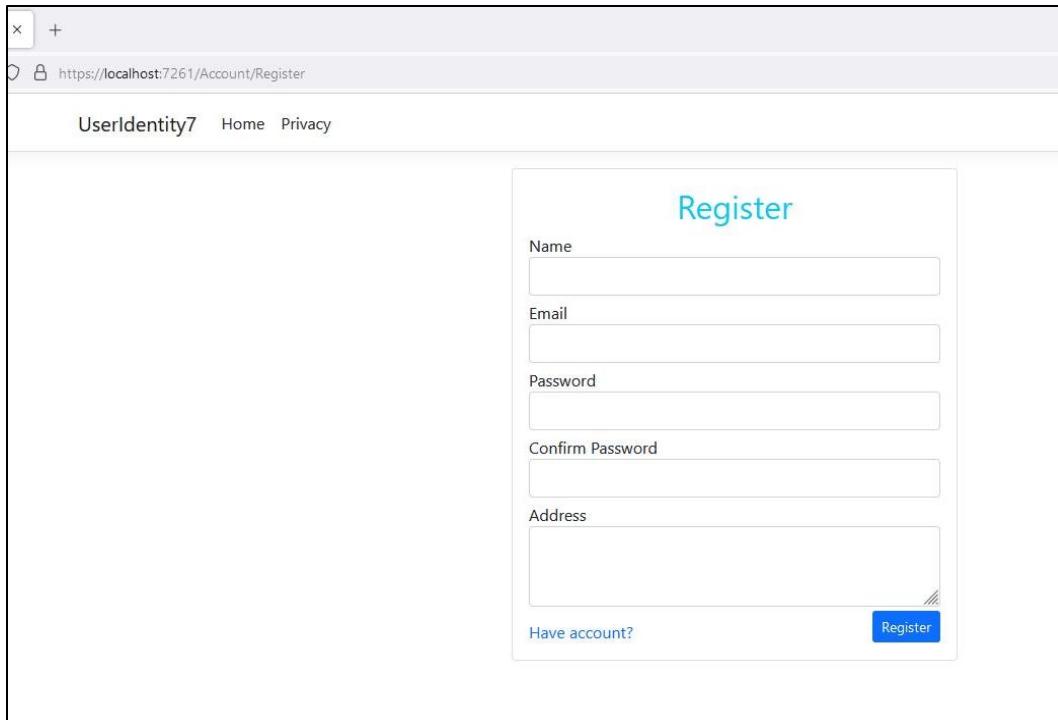
Build and run the application.

Final output in the browser is shown in Figures 14.6 and 14.7.



**Figure 14.6: Login View**

The applications opens with the Login page, however, for a first time user, registration is required. Click Register in the top corner of the browser window. The Register page is displayed as shown in Figure 14.7.



**Figure 14.7: Register View**

**Note:** If you get an `InvalidOperationException`: The view `Register` was not found error, navigate to your project solution in Solution Explorer, right-click and add the nuget package

`Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`. Then, add following bolded statement in `Program.cs`:

```
builder.Services.AddControllersWithViews() .AddRazorRuntimeCompilation();
```

Run the application again and the error will be resolved.

# Summary

- ✓ User identity and login are essential components of Web applications, allowing users to authenticate and access personalized features.
- ✓ ASP.NET Core provides built-in support for user authentication and identity management through ASP.NET Core Identity, simplifying the implementation of user authentication features.
- ✓ Best practices for password security include using strong and complex passwords, hashing passwords before storing them, and implementing additional security measures such as Multi-Factor Authentication (MFA).
- ✓ Session state management is crucial for maintaining user-specific data across multiple HTTP requests within the same session.
- ✓ Session options such as cookie properties and timeout period can be configured to meet the application's requirements.
- ✓ By implementing these features, developers can understand how to leverage ASP.NET Core Identity for user authentication and enhance security in their applications.

# Test Your Knowledge



1. What is ASP.NET Core Identity used for?

<b>A</b>	Handling session state management
<b>B</b>	Managing user authentication and identity
<b>C</b>	Implementing password encryption
<b>D</b>	Managing database connections
2. Which of the following is a recommended practice for ensuring password security?

<b>A</b>	Storing passwords in plaintext
<b>B</b>	Using weak and easily guessable passwords
<b>C</b>	Hashing passwords before storing them
<b>D</b>	Sharing passwords via email
3. Which middleware is used to enable session state in ASP.NET Core applications?

<b>A</b>	UseRouting
<b>B</b>	UseSession
<b>C</b>	UseAuthentication
<b>D</b>	UseStaticFiles
4. What is the purpose of session state management in Web applications?

<b>A</b>	To store sensitive user information
<b>B</b>	To authenticate users
<b>C</b>	To maintain user-specific data across requests within the same session
<b>D</b>	To encrypt communication between client and server
5. Which of the following is a common functionality provided by ASP.NET Core Identity?

<b>A</b>	File upload management
<b>B</b>	Email sending functionality
<b>C</b>	User authentication and authorization
<b>D</b>	Database query optimization

## Answers

1	B
2	C
3	B
4	C
5	C

## **Try It Yourself**

1. Create a new ASP.NET Core project and integrate ASP.NET Core Identity. Implement user registration functionality to allow users to sign up with a username and password. Implement user login functionality to authenticate users with their credentials. Test the registration and login process to ensure users can register, login, and access authenticated features.
2. Implement password policy settings in ASP.NET Core Identity to enforce strong password requirements (for example, minimum length, required characters). Implement password hashing and salting to securely store user passwords in the database. Test the password security implementation by registering users with different password combinations and validating the stored password hashes.

# *Session 15: Publishing and Deploying ASP.NET Core Applications*

## **Session Overview**

This session provides a comprehensive guide to the installation and deployment process of ASP.NET Core applications.

It covers essential configuration requirements and deployment procedures, including an introduction to Web servers, deployment options in Visual Studio 2022, and the installation of the .NET Core Hosting Bundle.

## **Objectives**

In this session, students will learn to:

- ✓ Describe Web Servers
- ✓ Explain the Deployment Process in Visual Studio 2022
- ✓ Describe IIS configuration
- ✓ Explain the process to deploy ASP.NET Core and ASP.NET MVC
- ✓ Explain the installation procedure of .NET Core Hosting Bundle

## **15.1 Introduction to Web Servers**

Web servers are software applications or hardware devices responsible for serving Web content to users over the Internet. They act as intermediaries between client devices, such as Web browsers and the backend systems hosting the Web applications.

### **15.1.1 What are Web Servers?**

A Web server is a specialized software application that processes incoming HTTP requests from clients (such as Web browsers) and delivers Web content (such as Web pages, images, videos, and so on.) in response. It runs on a physical server machine or virtualized environment and listens for incoming requests on designated ports, typically port 80 for HTTP and port 443 for HTTPS.

### **15.1.2 Role of Web Servers in Hosting ASP.NET Core Applications**

In the context of hosting ASP.NET Core applications, Web servers play a crucial role in processing incoming HTTP requests, routing them to the appropriate handlers within the ASP.NET Core application, and delivering the generated responses back to the clients.

ASP.NET Core applications can be hosted on different Web servers, each offering specific features and capabilities. These servers provide the necessary runtime

environment and services for executing ASP.NET Core applications, handling HTTP traffic, managing security, and enabling scalability.

Web servers employ various technologies and configurations to handle incoming requests efficiently, manage resources, and ensure the security and reliability of Web applications.

### 15.1.3 Commonly Used Web Servers

Several Web servers are commonly used for hosting ASP.NET Core applications. Some of the prominent ones include:

- **Internet Information Services (IIS):** Developed by Microsoft, IIS is one of the most widely used Web servers on the Windows platform. It provides robust features for hosting ASP.NET Core applications, including integration with the Windows operating system, support for various protocols and security features, and scalability options.
- **Apache HTTP Server:** Apache is an open-source Web server widely used on Unix-based operating systems (such as Linux). It supports multiple programming languages and frameworks, including ASP.NET Core through additional modules such as mod\_aspdotnet.
- **Nginx:** Nginx is a lightweight and high-performance Web server known for its efficiency in handling concurrent connections and serving static content. While commonly used as a reverse proxy or load balancer, Nginx can also host ASP.NET Core applications directly using technologies such as ASP.NET Core Module for Nginx (ASP.NET Core hosting bundle). This is typically used for Linux, rather than Windows.

## 15.2 Deployment Process in Visual Studio 2022

Visual Studio 2022 provides comprehensive tools and features to streamline the deployment of ASP.NET Core applications. Whether deploying to a local development environment, a production server, or a cloud platform such as Azure, Visual Studio offers various options to simplify the deployment process.

### 15.2.1 Overview of Deployment Options in Visual Studio 2022

Following are multiple deployment options offered by Visual Studio 2022 to cater to different deployment scenarios and requirements:

1. **Publish to Folder:** This option allows developers to publish their ASP.NET Core application to a local directory on the development machine. They can then manually copy the published files to a Web server or hosting environment.
2. **Publish to IIS:** Visual Studio integrates seamlessly with Internet Information Services (IIS), allowing developers to publish their ASP.NET Core application directly to a local IIS server for testing and development purposes.

3. **Publish to Azure:** Visual Studio provides built-in support for deploying ASP.NET Core applications to Microsoft Azure App Service, Azure Virtual Machines, Azure Kubernetes Service (AKS), and other Azure hosting options. These are part of Microsoft Azure cloud platform products. Developers can deploy directly from Visual Studio to Azure, simplifying the process of hosting their application in the cloud.

### 15.2.2 Publishing Profiles and Configurations

Visual Studio allows developers to create publishing profiles to define the deployment settings and configurations for their ASP.NET Core application. Publishing profiles store information such as the target destination, connection settings, deployment options, and any additional parameters required for deployment.

Key aspects of publishing profiles include:

- **Target Environment:** Specify the target environment for deployment, such as a local folder, IIS server, or Azure App Service.
- **Connection Settings:** Provide connection details for remote servers or cloud platforms, including credentials and authentication methods.
- **Deployment Options:** Configure deployment settings such as file system layout, pre/post-deployment actions, and optimization options.
- **Additional Parameters:** Customize deployment behavior by specifying additional parameters or command-line arguments as required.

By creating and managing publishing profiles, developers can easily switch between different deployment configurations and streamline the deployment process for their ASP.NET Core applications.

### 15.2.3 Deployment to Local IIS or Azure App Service

Visual Studio simplifies the deployment of ASP.NET Core applications to local IIS servers and Microsoft Azure App Service:

- **Local IIS Deployment:** With Visual Studio, developers can publish their ASP.NET Core application directly to a local IIS server for testing and debugging. The deployment process configures IIS to host their application and ensures seamless integration with the development environment.
- **Azure App Service Deployment:** Visual Studio provides seamless integration with Microsoft Azure, allowing developers to deploy ASP.NET Core applications to Azure App Service with minimal effort. Developers can choose from various deployment options, including Continuous Integration (CI) pipelines, deployment slots, and scalability features offered by Azure App Service.

### 15.3 Configuring ASP.NET Core App to IIS

ASP.NET Core hosting requirements are different from ASP.NET because the configurations are different. IIS, a Web server that operates on the Windows OS ASP.NET framework, is generally used to host ASP.NET Core applications.

Every ASP.NET Core project is internally a console application that has a `Program.cs` file. The auto-generated code in the `Program.cs` file is as shown in Code Snippet 1.

#### Code Snippet 1

```
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();
```

To configure and create Web hosts, ASP.NET Core now uses the `WebApplication` class, which provides a streamlined approach to setting up ASP.NET Core applications. Instead of `WebHostBuilder`, developers use `WebApplication.CreateBuilder(args)` to initialize the application builder and configure services. The `UseKestrel()` method is still commonly used to configure Kestrel as the Web server, and `UseIISIntegration()` is used to enable integration with IIS.

## 15.4 Deploying ASP.NET Core to IIS

To deploy an ASP.NET Core application to IIS, the steps are as follows:

1. The first step is to configure IIS. To configure IIS, open **Control Panel** and then, click **Programs** as shown in Figure 15.1.

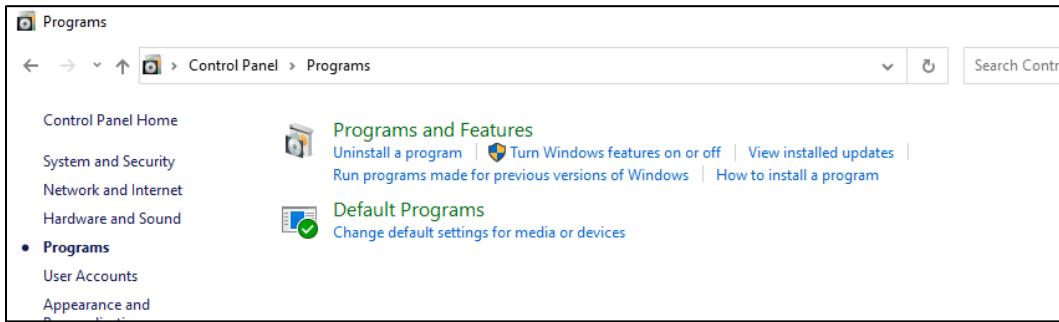


Figure 15.1: Control Panel

2. Next, click the **Turn Windows Features on or off** and select **Internet Information Services** and all its hierarchy options. Click **OK** as shown in Figure 15.2.

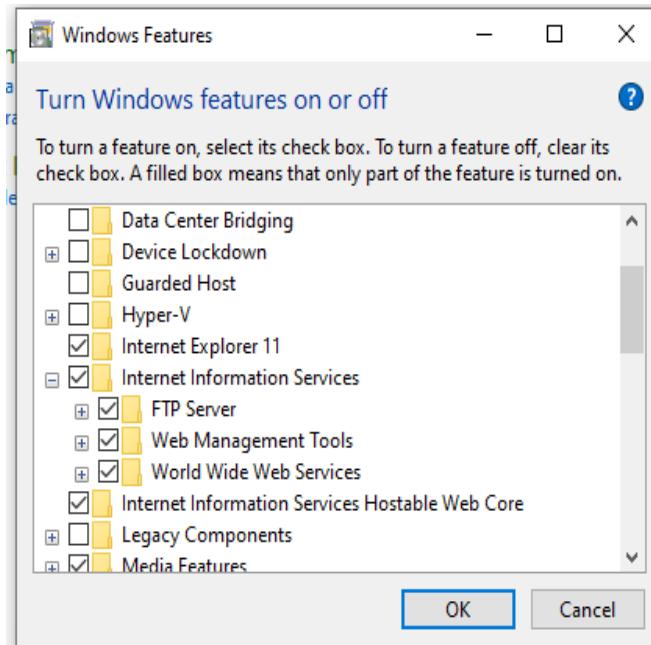
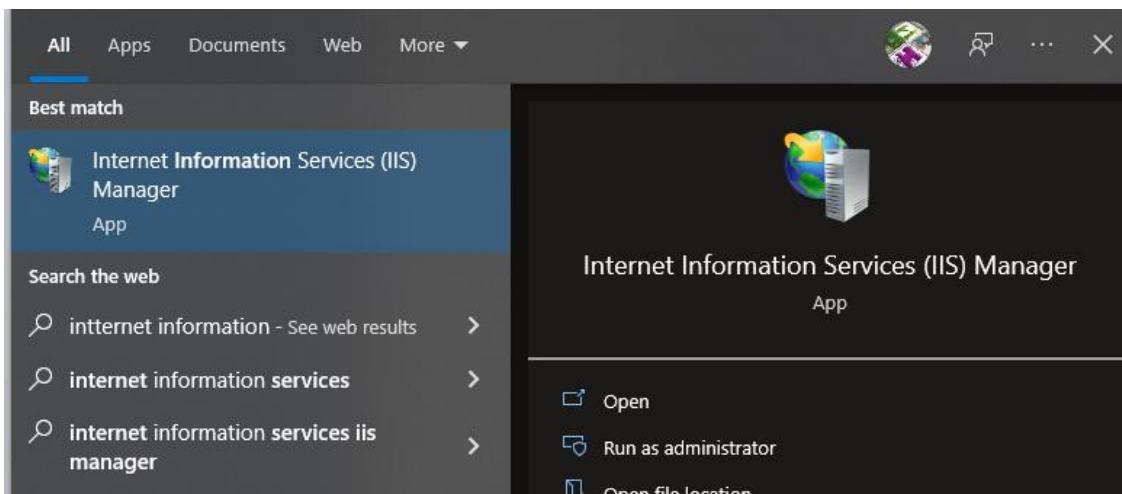


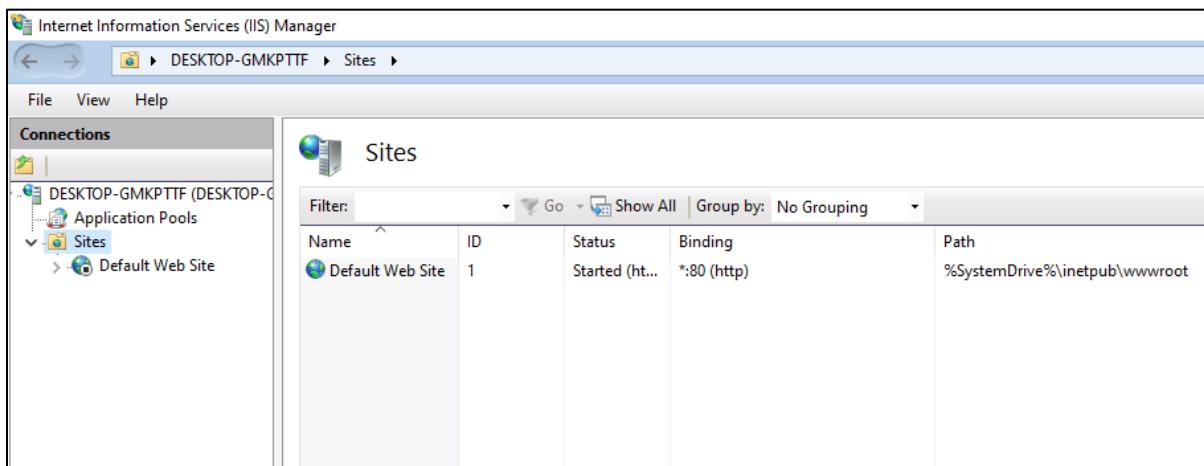
Figure 15.2: Turn Windows Features On or Off

3. To start the **Internet Information Services (IIS) Manager** App, click **Open** as shown in Figure 15.3.



**Figure 15.3: IIS App**

4. In the **Internet Information Services (IIS) Manager** window, to provide information about the Website, click **Sites** as shown in Figure 15.4.



**Figure 15.4: Sites in IIS Manager**

5. Next, provide information about the Website in the **Add Website** window. Provide the Site Name, Application Pool, and path of the Website. Figure 15.5 shows creation of folder in **wwwroot** for physical location of the Website. Also, provide the Host Name. This is done so that during deployment, application can be opened using Host Name without the requirement of an IP address. Refer to Figure 15.6.

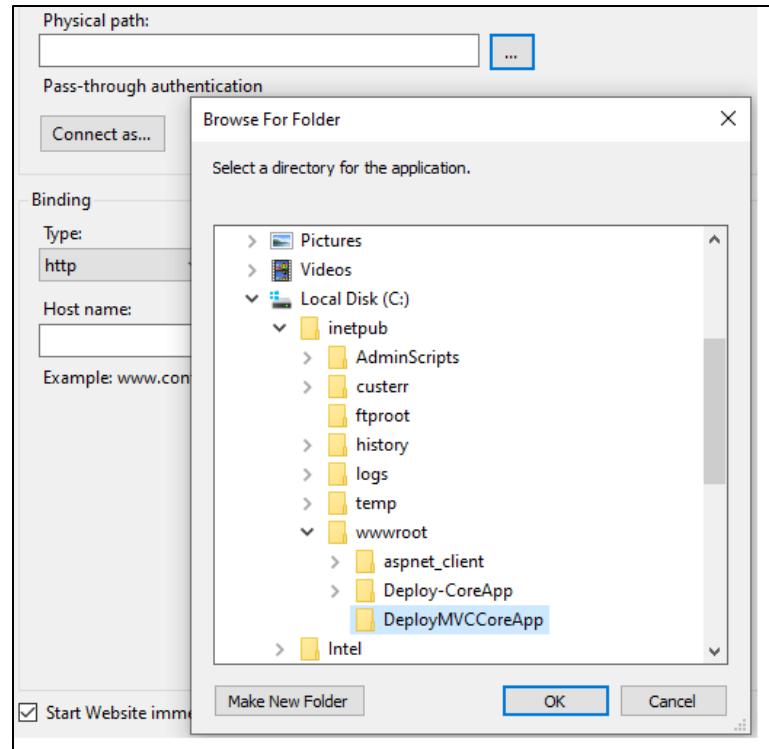


Figure 15.5: Physical Path

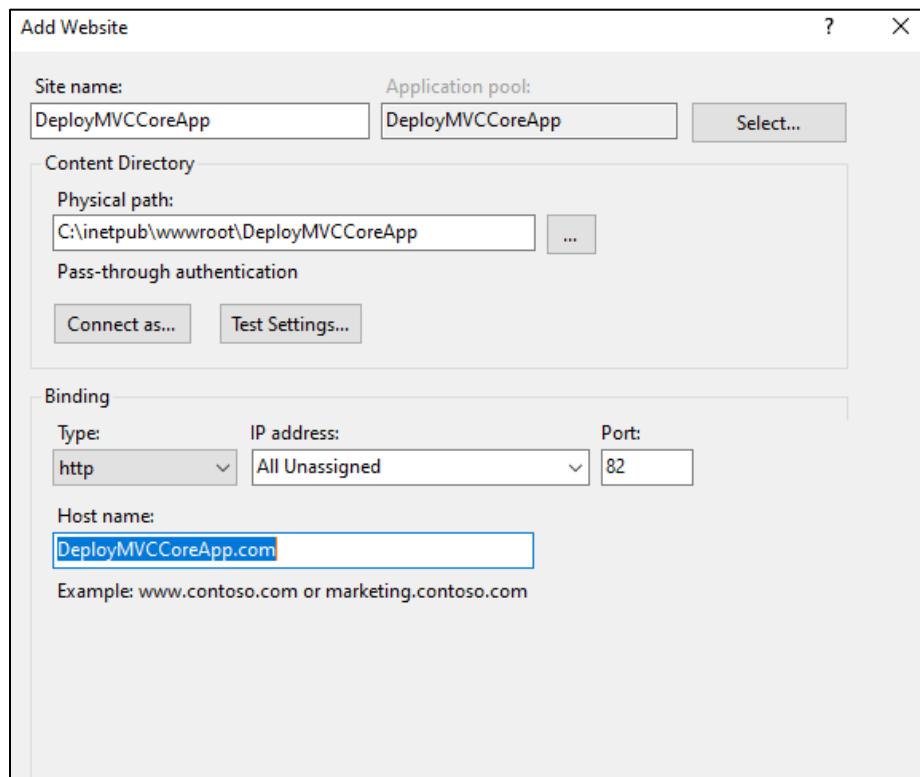
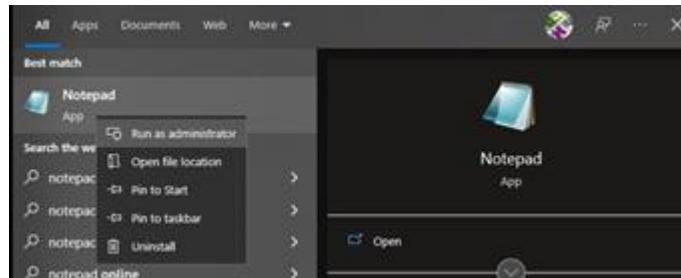


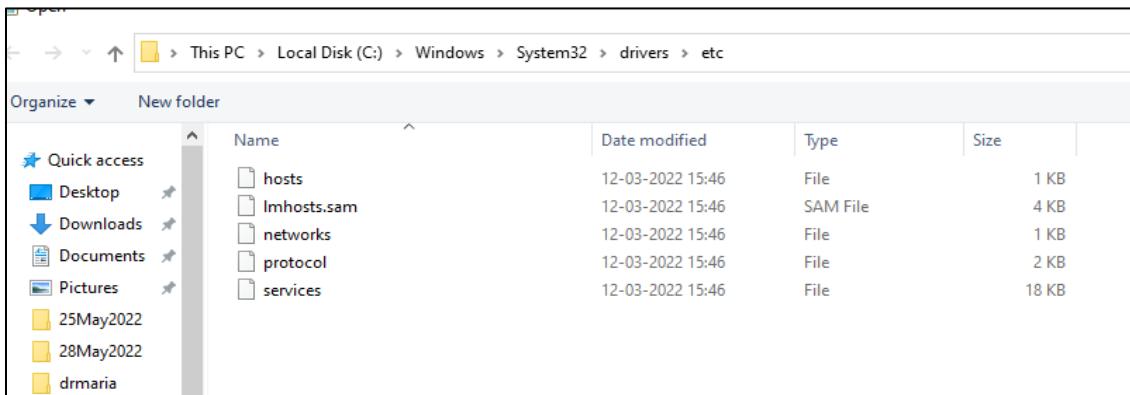
Figure 15.6: Add Website

6. Next step is to modify the **hosts** file for the system. Open **Notepad** and click **Run as administrator** to open it in the Administrator mode as shown in Figure 15.7.



**Figure 15.7: Open Notepad in Administrator Mode**

The *hosts* file is in the path **Windows → System 32 → drivers → etc**. Refer to Figure 15.8.



**Figure 15.8: Hosts File**

7. Open the file through Notepad in Administrator mode. Unless this is done, the file cannot be saved after editing.

The *hosts* file is opened and the content of the file is as shown in Figure 15.9.

```
*hosts - Notepad
File Edit Format View Help
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97      rhino.acme.com      # source server
#      38.25.63.10      x.acme.com          # x client host
#
# localhost name resolution is handled within DNS itself.
#      127.0.0.1      localhost
#      ::1            localhost|
```

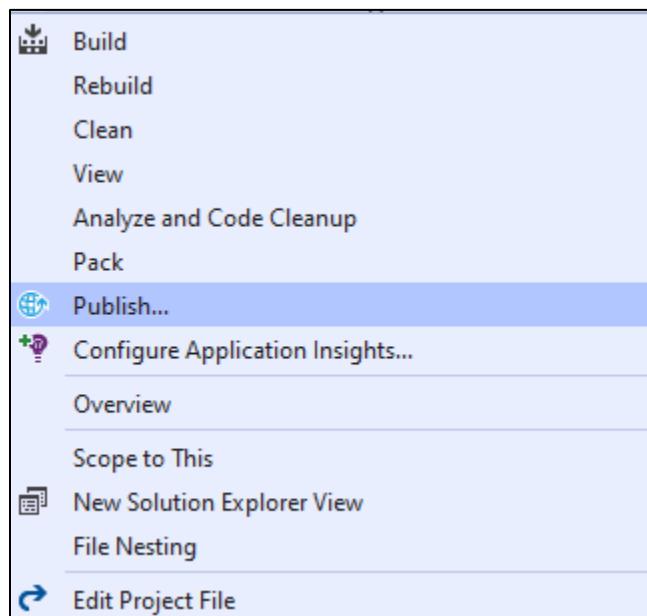
**Figure 15.9: Host File Content**

8. Finally, to map the localhost to **DeployMVCCoreApp.com**, in the hosts file, add the line **127.0.0.1 DeployMVCCoreApp.com** as shown in Figure 15.10.

```
# For example:  
#  
#      102.54.94.97      rhino.acme.com  
#      38.25.63.10      x.acme.com  
  
# localhost name resolution is handled within  
#      127.0.0.1      localhost  
#      ::1            localhost  
  
127.0.0.1      DeployMVCCoreApp.com
```

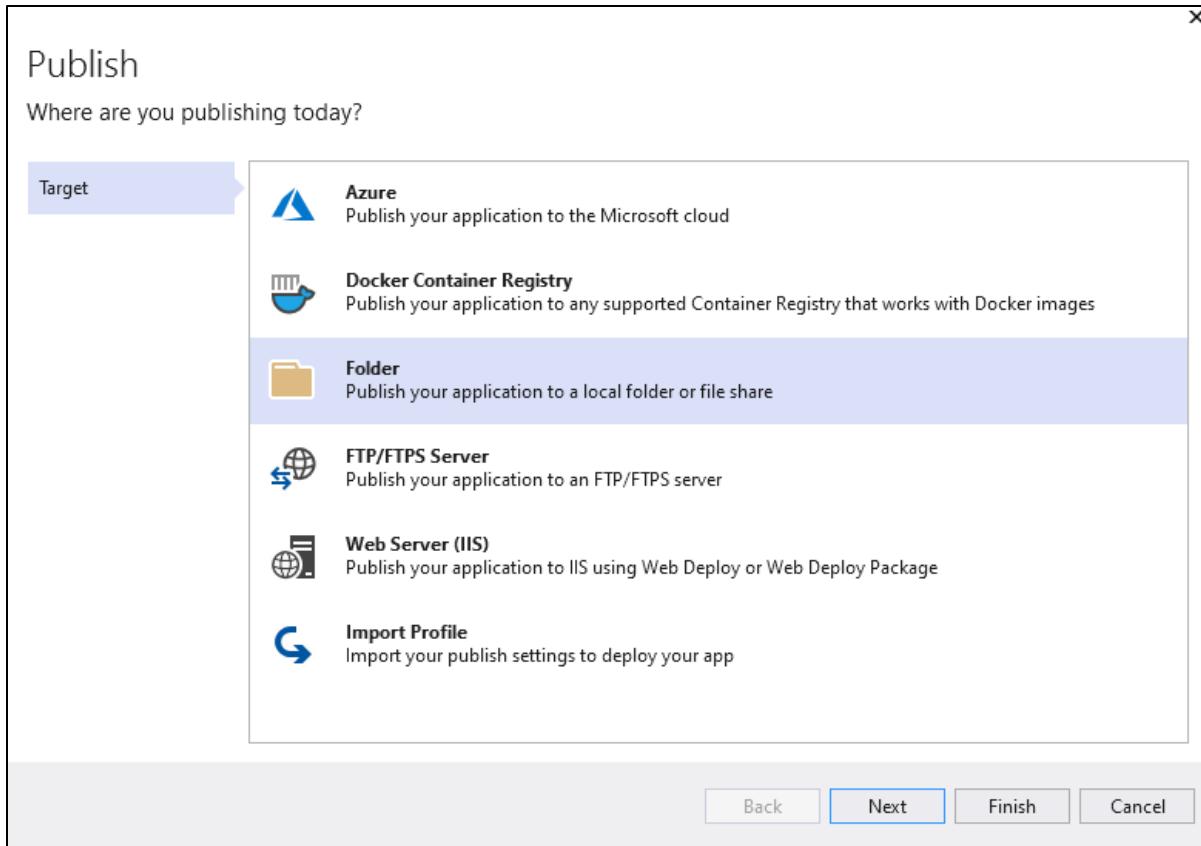
**Figure 15.10: hosts File**

9. Next, to publish the application, open the application in Visual Studio 2022 IDE. Click **Publish** as shown in Figure 15.11.

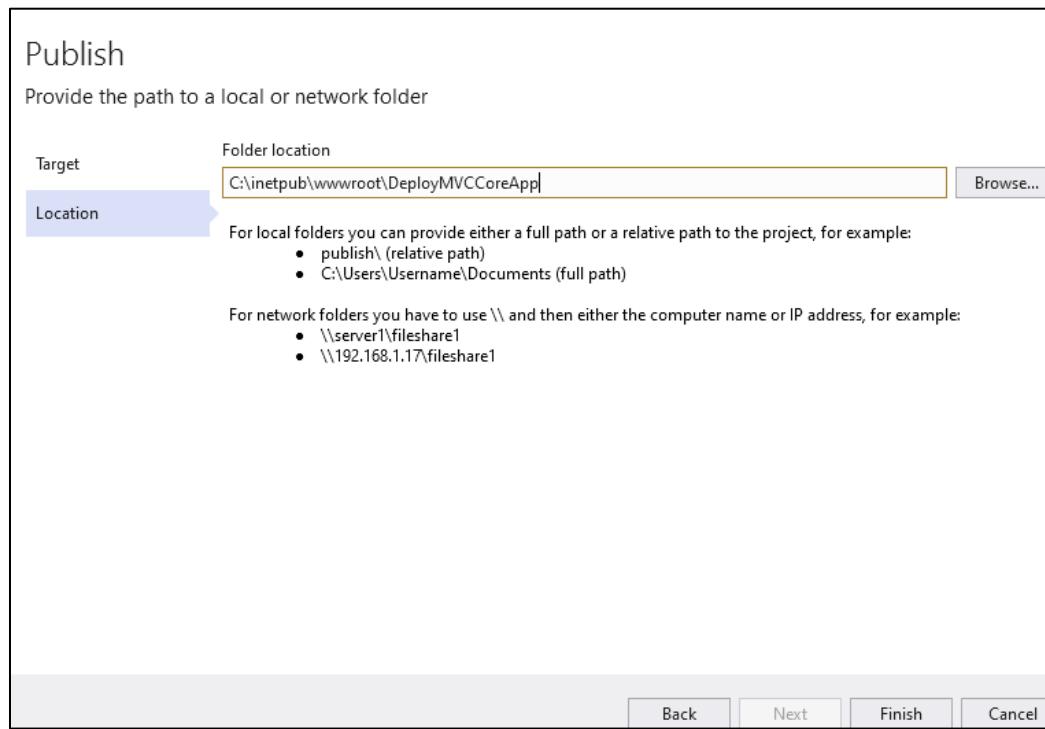


**Figure 15.11: Publish**

10. To select the target for publishing or deploying the application, there are various options. Azure allows publishing the applications to Microsoft Cloud. The application can be published directly to IIS using Web Deploy. In this example, application is deployed in IIS through creation of the folder **DeployMVCCoreApp** that is in the **wwwroot** folder. This is done to connect IIS configuration with Visual Studio Publishing configuration. Select **Folder** and then, provide the path in **Folder location** as shown in Figures 15.12 and 15.13.

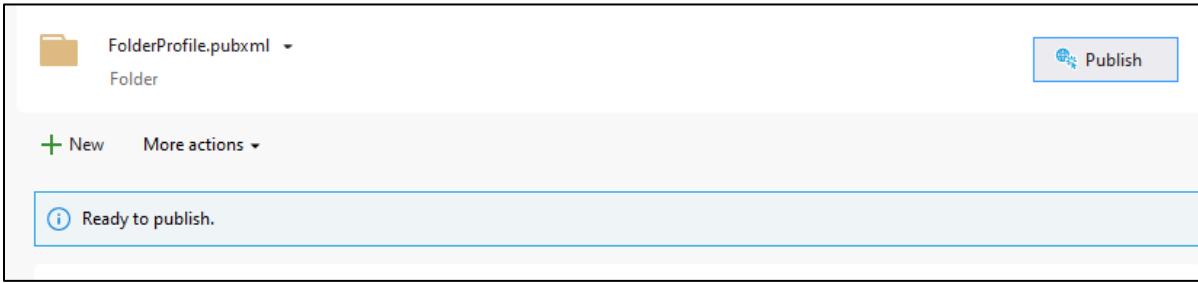


**Figure 15.12: Deployment**



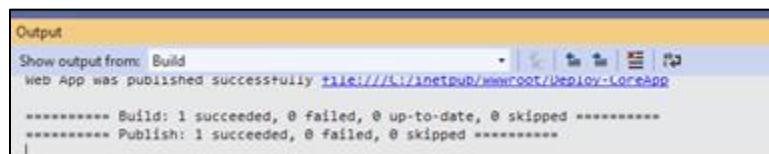
**Figure 15.13: Folder Location**

11. Finally, click **Publish** as shown in Figure 15.14 to publish the entire Web application at the IIS location.



**Figure 15.14: Publish**

The Build succeeded message is displayed in the Output window as shown in Figure 15.15.



**Figure 15.15: Output**

12. Next, open the **wwwroot** location to view the application files that are created as shown in Figure 15.16.

This PC > Local Disk (C:) > inetpub > wwwroot > DeployMVCCoreApp			
Name	Date modified	Type	Size
wwwroot	29-05-2022 17:30	File folder	
appsettings.Development	29-05-2022 16:41	JSON File	1 KB
appsettings	29-05-2022 16:41	JSON File	1 KB
DeployMVCCoreApp.deps	29-05-2022 17:30	JSON File	105 KB
DeployMVCCoreApp.dll	29-05-2022 17:30	Application exten...	9 KB
DeployMVCCoreApp	29-05-2022 17:30	Application	171 KB
DeployMVCCoreApp.pdb	29-05-2022 17:30	Program Debug D...	20 KB
DeployMVCCoreApp.runtimeconfig	29-05-2022 17:01	JSON File	1 KB
DeployMVCCoreApp.Views.dll	29-05-2022 17:30	Application exten...	35 KB
DeployMVCCoreApp.Views.pdb	29-05-2022 17:30	Program Debug D...	21 KB
web.config	29-05-2022 17:30	XML Configuratio...	1 KB

**Figure 15.16: wwwroot Folder**

13. To open the application in browser via IIS, select the application and right-click to select **Manage Website** → **Browse**. Refer to Figure 15.17.

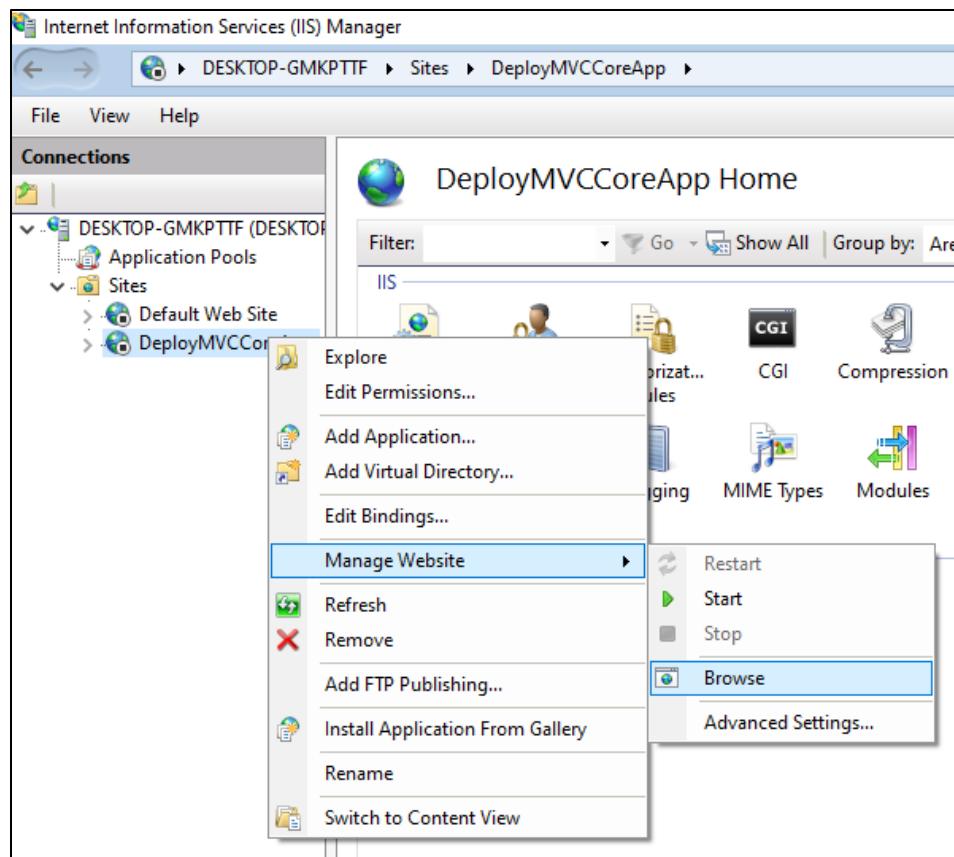


Figure 15.17: Manage Website

On clicking **Browse**, the **HTTP Error 500.19 – Internal Server Error** screen is displayed as shown in Figure 15.18.

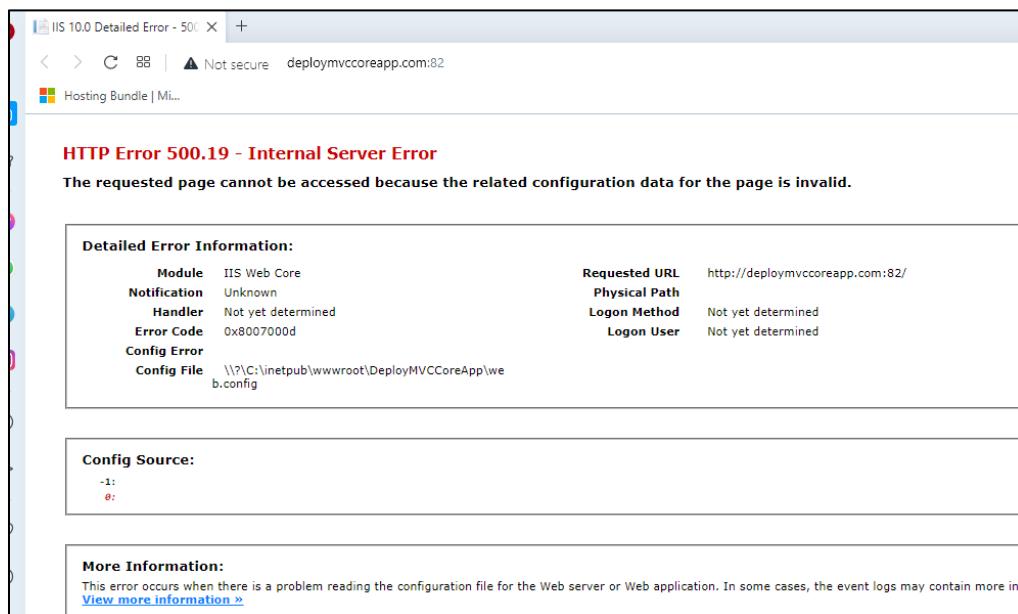
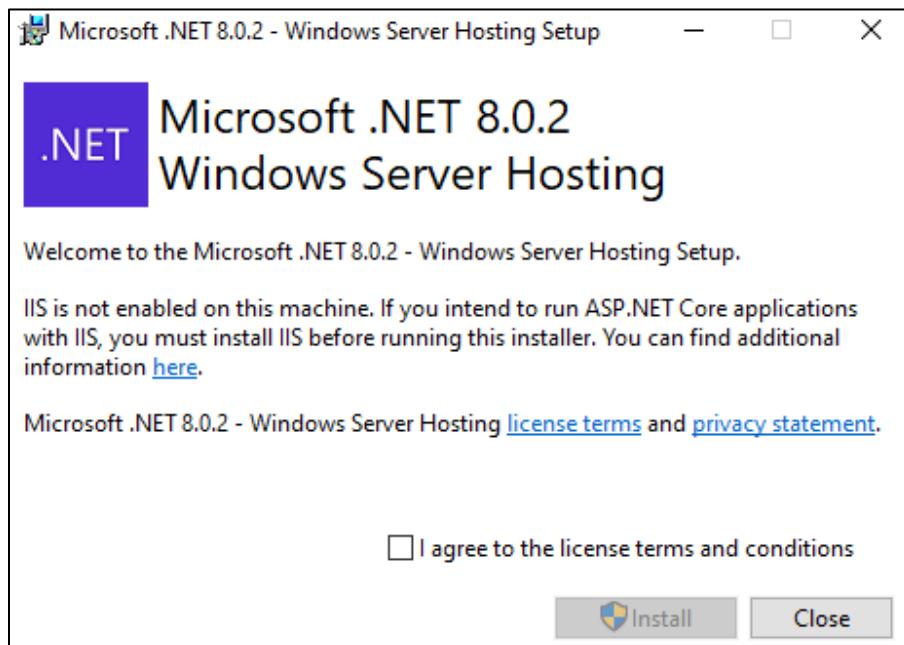


Figure 15.18: HTTP Error

This error is displayed because machine is not configured with Hosting Bundle Services. Therefore, open the URL <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/hosting-bundle?view=aspnetcore-6.0> and install the package as shown in Figures 15.19 and 15.20.

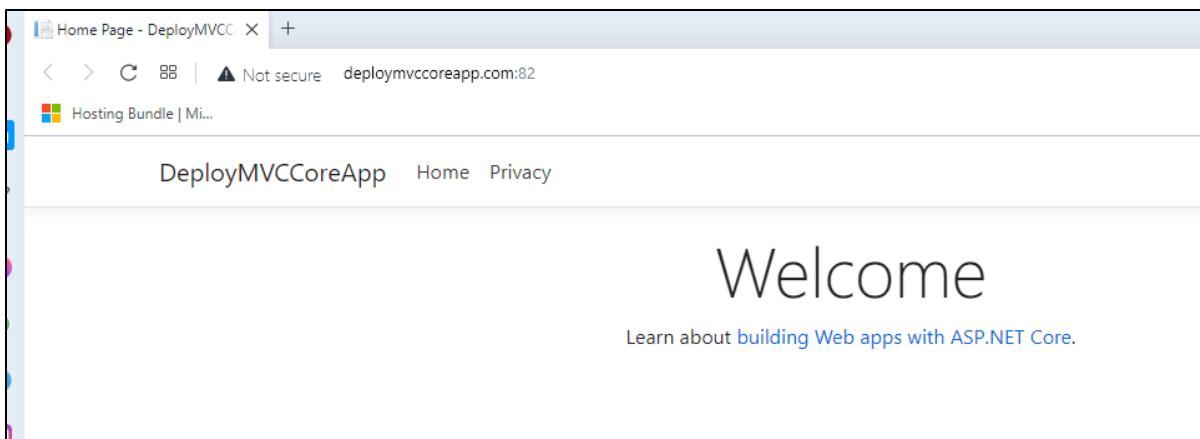


**Figure 15.19: Hosting Bundle Installer**



**Figure 15.20: Windows Server Hosting**

14. Next, repeat the steps for **Browse** after installation of Windows Server Hosting. **Welcome** screen of the application is displayed as shown in Figure 15.21.



**Figure 15.21: Welcome Screen**

**Note:** Deploying ASP.NET MVC can be done by following the same steps.

## 15.5 Installation Procedure for Windows features and .NET Core Hosting Bundle

The .NET Core Hosting Bundle is a package provided by Microsoft that includes the .NET Core runtime, ASP.NET Core runtime, and the ASP.NET Core Module for IIS. It is essential for hosting ASP.NET Core applications on Windows servers, as it provides the necessary components and runtimes required for running ASP.NET Core applications within the Internet Information Services (IIS) environment.

### Introduction to .NET Core Hosting Bundle

The .NET Core Hosting Bundle is designed to simplify the deployment and hosting of ASP.NET Core applications on Windows servers. It includes following components:

1. **.NET Core Runtime:** The .NET Core runtime is a cross-platform runtime environment that enables the execution of .NET Core applications. It provides necessary libraries, frameworks, and runtime components for running ASP.NET Core applications.
2. **ASP.NET Core Runtime:** The ASP.NET Core runtime is a runtime environment specifically for ASP.NET Core applications. It includes the ASP.NET Core libraries and frameworks required for processing HTTP requests, handling routing, managing sessions, and other Web application functionalities.
3. **ASP.NET Core Module for IIS:** The ASP.NET Core Module is an IIS module that integrates ASP.NET Core applications with the IIS Web server. It provides features such as request routing, process management, and integration with the IIS pipeline, enabling seamless hosting of ASP.NET Core applications on IIS.

## Purpose and Benefits of the Hosting Bundle

The .NET Core Hosting Bundle serves several purposes and offers several benefits for hosting ASP.NET Core applications:

- **Simplified Deployment:** By bundling the necessary runtime components, the hosting bundle simplifies the deployment process for ASP.NET Core applications on Windows servers. Developers can deploy applications with confidence, knowing that the required runtime components are included.
- **Improved Performance:** The hosting bundle includes optimized runtime components tailored for ASP.NET Core applications, resulting in improved performance and efficiency compared to standalone installations.
- **Seamless Integration with IIS:** The ASP.NET Core Module for IIS enables seamless integration between ASP.NET Core applications and the IIS Web server. It handles tasks such as request routing, process management, and configuration, ensuring smooth operation of ASP.NET Core applications within the IIS environment.

## Step-by-Step Installation Procedure on Windows

Follow these steps to install the .NET Core Hosting Bundle on a Windows server:

1. **Download the Hosting Bundle:** Visit the official Microsoft Website (<https://dotnet.microsoft.com/download/dotnet>) to download the .NET Core Hosting Bundle suitable for your Windows server version.
2. **Run the Installer:** Once the download is complete, run the installer executable (.exe) file to start the installation process.
3. **Accept License Terms:** Read and accept the license terms and agreements presented during the installation process.
4. **Choose Installation Options:** Select the installation options as per requirements. Typically, developers will want to install all available components, including the .NET Core runtime, ASP.NET Core runtime, and ASP.NET Core Module for IIS.
5. **Complete Installation:** Follow the on-screen prompts to complete the installation process. The installer will copy the necessary files, configure system settings, and register components as required.
6. **Restart Server:** After the installation is complete, restart the server to apply the changes and ensure that the installed components are activated.

## Verifying Installation and Troubleshooting Common Issues

To verify the installation of the .NET Core Hosting Bundle and troubleshoot common issues, developers should follow these steps:

1. **Check Installed Components:** Confirm that the .NET Core runtime, ASP.NET Core runtime, and ASP.NET Core Module for IIS are installed correctly by checking the installed programs list in Windows or using command-line tools such as `dotnet --info`.
2. **Test ASP.NET Core Applications:** Deploy a sample ASP.NET Core application to Windows server and verify that it runs successfully within the IIS environment.

Ensure that all functionality works as expected and that there are no runtime errors or issues.

3. **Review Event Logs:** Check the Windows Event Viewer for any error messages or warnings related to the installation or operation of the .NET Core Hosting Bundle. Event logs can provide valuable insights into potential issues or misconfigurations.
4. **Consult Documentation and Support:** Refer to the official Microsoft documentation and support resources for guidance on troubleshooting specific issues or configuration problems related to the .NET Core Hosting Bundle. Online forums, community discussions, and support channels can also provide assistance from experienced users and Microsoft support staff.

# Summary

- ✓ Web server is software or hardware that handles incoming HTTP requests from clients (such as Web browsers) and serves Web content in response.
- ✓ Visual Studio 2022 offers multiple deployment options, including publishing to a local folder, local IIS server, or Azure App Service.
- ✓ ASP.NET Core hosting requirements are different from ASP.NET as its configurations are different.
- ✓ ASP.NET Core project is internally a console application that has a `Program.cs` file.
- ✓ ASP.NET Core uses `WebApplication` to configure and create `WebHosts`.
- ✓ To select the target for publishing or deploying the application, there are various options.
- ✓ Azure allows publishing the applications to Microsoft Cloud.
- ✓ Applications can be published directly to IIS using Web Deploy.
- ✓ The .NET Core Hosting Bundle simplifies the deployment and hosting of ASP.NET Core applications on Windows servers.

# Test Your Knowledge



1. Which of the following options can be used to deploy an application on IIS Web Server?

<b>A</b>	Azure
<b>B</b>	IIS
<b>C</b>	Web Deploy
<b>D</b>	Command line

2. To configure and create Web hosts, ASP.NET Core uses \_\_\_\_\_.

<b>A</b>	WebApplication
<b>B</b>	UseKestrel
<b>C</b>	UseIISIntegration
<b>D</b>	None of these

3. ASP.NET Core projects create a \_\_\_\_\_ file to deploy an application to IIS and register the `AspNetCoreModule` as an HTTP handler.

<b>A</b>	Web.Host
<b>B</b>	Web.config
<b>C</b>	IISIntegration
<b>D</b>	Json

4. Which of the following Web server that operates on the Windows OS ASP.NET framework is used to host ASP.NET Core apps?

<b>A</b>	IIS
<b>B</b>	Apache Tomcat
<b>C</b>	JBoss
<b>D</b>	None of these

5. Which of the following options are used to publish an ASP.NET Core/MVC application on Microsoft Cloud?

<b>A</b>	Azure
<b>B</b>	IIS
<b>C</b>	Web Deploy
<b>D</b>	All of these

## Answers

1	C
2	A
3	B
4	A
5	A

## **Try It Yourself**

1. Create a new ASP.NET Core application and deploy to a local Internet Information Services (IIS) server using Visual Studio 2022.
2. Create a new ASP.NET Core application and deploy to Microsoft Azure App Service using Visual Studio 2022.

# Appendix

Sr. No.	Case Studies
1.	<p><b>Project 1: Building a Task Management Web Application</b></p> <p>Scenario: The goal of this exercise is to design and implement a Web-based Task Management System that allows users within a company to efficiently track projects, tasks, and deadlines. The system should provide features for task creation, updating, and deletion, as well as project management capabilities. Additionally, the application should include user authentication, Role-Based Access Control (RBAC), and a dashboard to view tasks and projects.</p> <p><b>Functional Requirements:</b></p> <ol style="list-style-type: none"> <li>1. <b>User Authentication:</b> <ul style="list-style-type: none"> <li>○ Users should be able to register for an account or log in with existing credentials.</li> <li>○ Authentication should be implemented securely to protect user data.</li> </ul> </li> <li>2. <b>RRBAC:</b> <ul style="list-style-type: none"> <li>○ The system should have different user roles such as Admin, Manager, and Employee.</li> <li>○ Admins should have full access to all features and functionalities.</li> <li>○ Managers should be able to create, update, and delete projects and assign tasks to employees.</li> <li>○ Employees should only be able to view tasks assigned to them and update their task status.</li> </ul> </li> <li>3. <b>Task Management:</b> <ul style="list-style-type: none"> <li>○ Users should be able to create new tasks, specifying details such as task name, description, priority, deadline, and assigned project.</li> <li>○ Tasks should be editable, allowing users to update task details or mark tasks as complete.</li> <li>○ Users should have the option to delete tasks, if they are no longer relevant.</li> </ul> </li> <li>4. <b>Project Management:</b> <ul style="list-style-type: none"> <li>○ Admins and managers should be able to create new projects and assign tasks to specific projects.</li> <li>○ Projects should have a name, description, start date, and end date.</li> </ul> </li> <li>5. <b>Dashboard:</b> <ul style="list-style-type: none"> <li>○ Upon logging in, users should be presented with a dashboard displaying an overview of their tasks and projects.</li> <li>○ The dashboard should include summary statistics such as total tasks assigned, tasks completed, upcoming deadlines, and so on.</li> </ul> </li> </ol>

	<p><b>Technical Requirements:</b></p> <ol style="list-style-type: none"> <li>1. <b>ASP.NET MVC or ASP.NET Core:</b> <ul style="list-style-type: none"> <li>○ Choose either ASP.NET MVC or ASP.NET Core as the framework for developing the Web application. Use Visual Studio 2022 as the IDE.</li> </ul> </li> <li>2. <b>Database Integration:</b> <ul style="list-style-type: none"> <li>○ Use Entity Framework Core to integrate with a SQL Server 2022 database for storing user accounts, tasks, projects, and other data.</li> </ul> </li> <li>3. <b>User Interface:</b> <ul style="list-style-type: none"> <li>○ Design an intuitive and user-friendly interface for creating, updating, and managing tasks and projects.</li> <li>○ Implement responsive design principles to ensure compatibility across different devices and screen sizes.</li> </ul> </li> <li>4. <b>Security Measures:</b> <ul style="list-style-type: none"> <li>○ Implement appropriate security measures such as input validation, authentication, authorization, and protection against common Web vulnerabilities (for example, SQL injection, cross-site scripting, and so on).</li> </ul> </li> <li>5. <b>Testing and Validation:</b> <ul style="list-style-type: none"> <li>○ Conduct thorough testing of the application to ensure functionality, usability, and security.</li> <li>○ Validate user inputs and handle edge cases gracefully to prevent errors and data inconsistencies.</li> </ul> </li> </ol>
2.	<p><b>Project 2: Creating a Blogging Platform</b></p> <p>Scenario: The objective of this exercise is to design and implement a feature-rich Blogging Platform that empowers content creators to share their articles, engage with readers through comments, and customize the appearance of their blog. The platform should provide a user-friendly interface for managing blog posts, comments, and user accounts, while also offering essential features such as user registration, authentication, and search functionality.</p> <p><b>Functional Requirements:</b></p> <ol style="list-style-type: none"> <li>1. <b>User Registration and Authentication:</b> <ul style="list-style-type: none"> <li>○ Users should be able to register for a new account by providing basic details such as username, email, and password.</li> <li>○ Registered users should be able to log in securely using their credentials.</li> <li>○ Authentication should be implemented to protect sensitive actions and data within the application.</li> </ul> </li> <li>2. <b>Blog Post Management:</b> <ul style="list-style-type: none"> <li>○ Content creators should be able to create new blog posts, including titles, content, categories, tags, and featured images.</li> <li>○ Posts should support rich text formatting, media embedding, and other multimedia elements.</li> </ul> </li> </ol>

- Users should have the ability to edit, delete, and publish/unpublish their blog posts.
- 3. Comment Management:**
- Readers should be able to leave comments on published blog posts.
  - Content creators should have the ability to moderate comments, including approval, deletion, and flagging/reporting functionality.
  - Comments should support basic formatting and allow for threaded discussions.
- 4. Customization and Theming:**
- Content creators should be able to customize the appearance of their blog, including layout, color scheme, fonts, and branding elements.
  - The platform should support multiple themes or templates for users to choose from.
- 5. User Interaction and Engagement:**
- Readers should have the ability to such as, share, and bookmark blog posts for future reference.
  - Content creators should be able to view analytics and statistics related to their blog posts, such as page views, comments, and engagement metrics.
- 6. Search Functionality:**
- Users should be able to search for blog posts by keywords, categories, tags, or author names.
  - Search results should be displayed in a user-friendly manner, with options for filtering and sorting.

### **Technical Requirements:**

- 1. ASP.NET Web Forms or ASP.NET Core MVC:**
  - Choose either ASP.NET MVC or ASP.NET Core as the framework for developing the Web application. Use Visual Studio 2022 as the IDE.
- 2. Database Integration:**
  - Use Entity Framework Core or other ORM frameworks to integrate with a SQL Server 2022 database for storing blog posts, comments, user accounts, and other data.
- 3. User Interface and Frontend Development:**
  - Design an intuitive and visually appealing user interface for navigating and interacting with blog posts, comments, and other content.
  - Implement responsive design principles to ensure compatibility across different devices and screen sizes.
- 4. Security Measures:**
  - Implement robust security measures to protect user data, including input validation, authentication, authorization, and protection against common Web vulnerabilities.

	<p><b>5. Testing and Validation:</b></p> <ul style="list-style-type: none"> <li>○ Conduct thorough testing of the application to ensure functionality, usability, and security.</li> <li>○ Validate user inputs and handle edge cases gracefully to prevent errors and data inconsistencies</li> </ul>
3.	<p><b>Project 3: E-commerce Website Development</b></p> <p><b>Scenario:</b> The goal of this exercise is to design and implement a fully functional E-commerce Website that enables a retail business to sell products online and provide a seamless shopping experience for customers. The platform should offer features for browsing products, adding items to the shopping cart, completing orders, and managing user accounts. Additionally, the application should include essential functionalities such as product categorization, search functionality, user authentication, and payment processing integration to facilitate online transactions.</p> <p><b>Functional Requirements:</b></p> <ol style="list-style-type: none"> <li><b>1. User Registration and Authentication:</b> <ul style="list-style-type: none"> <li>○ Users should be able to register for a new account or log in with existing credentials.</li> <li>○ Registered users should have access to features such as viewing order history, managing addresses, and updating account information.</li> <li>○ Authentication should be implemented securely to protect user data and transactions.</li> </ul> </li> <li><b>2. Product Management:</b> <ul style="list-style-type: none"> <li>○ The platform should support the display of various products, categorized into different product categories or departments.</li> <li>○ Each product should have details such as name, description, price, images, and availability status.</li> <li>○ Admin users should have the ability to add, edit, or remove products from the catalog.</li> </ul> </li> <li><b>3. Shopping Cart Functionality:</b> <ul style="list-style-type: none"> <li>○ Users should be able to add products to their shopping cart, update quantities, and remove items as required.</li> <li>○ The shopping cart should display a summary of selected items, including prices and total order amount.</li> </ul> </li> <li><b>4. Order Management:</b> <ul style="list-style-type: none"> <li>○ Users should be able to proceed to checkout to complete their orders securely.</li> <li>○ The checkout process should include steps for providing shipping information, selecting payment methods, and reviewing order details before finalizing the purchase.</li> <li>○ Users should receive order confirmation emails with details of their purchases.</li> </ul> </li> </ol>

- |  |  |
|--|--|
|  | <p><b>5. Product Search and Filtering:</b></p> <ul style="list-style-type: none"> <li>○ The platform should include search functionality to allow users to search for products by keywords, categories, or other criteria.</li> <li>○ Users should have the option to filter search results based on attributes such as price range, brand, or product features.</li> </ul> <p><b>6. Payment Processing Integration:</b></p> <ul style="list-style-type: none"> <li>○ The application should integrate with a payment gateway to securely process online payments.</li> <li>○ Supported payment methods may include credit/debit cards, PayPal, or other popular payment options.</li> </ul> |
|--|--|

**Technical Requirements:**

1. **ASP.NET Core MVC or ASP.NET Web Forms:**
  - Choose either ASP.NET Core MVC or ASP.NET Web Forms as the framework for developing the e-commerce platform. Use Visual Studio 2022 as the IDE.
2. **Database Integration:**
  - Use Entity Framework Core or other ORM frameworks to integrate with a SQL Server 2022 database for storing product data, user accounts, orders, and other relevant information.
3. **User Interface and Frontend Development:**
  - Design a user-friendly and visually appealing interface for browsing products, managing shopping carts, and completing orders.
  - Implement responsive design principles to ensure compatibility across different devices and screen sizes.
4. **Security Measures:**
  - Implement robust security measures to protect user data, transactions, and sensitive information.
  - Ensure compliance with industry standards and regulations regarding online payment processing and data protection.
5. **Testing and Validation:**
  - Conduct thorough testing of the application to ensure functionality, usability, and security.
  - Validate user inputs, handle edge cases gracefully, and perform stress testing to evaluate performance under load.

4.	<b>Project 4: Online Learning Management System</b>
----	---

**Scenario:**

The objective of this exercise is to design and develop a comprehensive Online Learning Management System (LMS) that enables educational institutions to deliver courses, assignments, and assessments to students remotely. The LMS should provide a user-friendly platform for course

creation, enrollment, content delivery, and student progress tracking, while also incorporating essential features such as user authentication, RBAC, discussion forums, and grading functionality to facilitate effective online learning.

#### **Functional Requirements:**

##### **1. User Authentication and RBAC:**

- Users should be able to register for an account or log in with existing credentials.
- The system should support different user roles such as students, instructors, and administrators.
- Instructors should have privileges to create and manage courses, assignments, and assessments.
- Students should have access to enrolled courses, assignments, and course materials.

##### **2. Course Management:**

- Instructors should be able to create new courses, specifying details such as course name, description, prerequisites, and course materials.
- Courses should support multimedia content such as videos, presentations, documents, and quizzes.
- Instructors should have the ability to organize course content into modules or sections for easy navigation.

##### **3. Enrollment and Student Management:**

- Students should be able to browse available courses, view course details, and enroll in courses of interest.
- Instructors should be able to manage student enrollments, track attendance, and monitor student progress.

##### **4. Content Delivery and Assessment:**

- The platform should support the delivery of course content, including lectures, readings, assignments, and assessments.
- Instructors should be able to create assignments, quizzes, and exams, and students should be able to submit their work online.
- The system should facilitate the grading and feedback process, allowing instructors to provide feedback and scores to students.

##### **5. Discussion Forums and Collaboration:**

- Each course should have a dedicated discussion forum where students can ask questions, discuss topics, and collaborate with peers and instructors.
- Instructors should be able to moderate discussions, provide answers, and encourage participation.

**Technical Requirements:****1. ASP.NET MVC or ASP.NET Core:**

- Choose either ASP.NET MVC or ASP.NET Core as the framework for developing Online Learning Management System. Use Visual Studio 2022 as the IDE.

**2. Database Integration:**

- Use Entity Framework Core or other ORM frameworks to integrate with a SQL Server 2022 database for storing course data, user accounts, assignments, assessments, and other relevant information.

**3. User Interface and Frontend Development:**

- Design an intuitive and user-friendly interface for navigating courses, accessing course materials, and interacting with assignments and assessments.
- Implement responsive design principles to ensure compatibility across different devices and screen sizes.

**4. Security Measures:**

- Implement robust security measures to protect user data, course content, and sensitive information.
- Ensure compliance with privacy regulations and standards regarding student data and online learning platforms.

**5. Testing and Validation:**

- Conduct thorough testing of the application to ensure functionality, usability, and security.
- Validate user inputs, handle edge cases gracefully, and perform stress testing to evaluate performance under load.