.NET CODING STANDARDS AND BEST PRACTICES

Version	Date	Reason for Change
1.0	01-Aug-2016	Base line and Approved By Management

Table of Contents

1. Naming Conventions and Standards

1.1. CamelCase

"Camel case" is the practice of writing identifiers in which no underscores are used to separate words, the first letter is lowercase, and the first letter of each subsequent word is capitalized.

Examples: fileName, voterAddress. Use camel case to name private and protected fields, parameters, and local variables.

1.2. PascalCase

In "Pascal case", no underscores are used to separate words, and the first character of all words is in upper case and other characters are in lower case.

Examples: GetFileName, MainForm. Use Pascal case for public fields and properties, class names including enumerated types and structures), and namespaces.

1.3. NamingConstants

All constants should be named in uppercase with underscores to separate words.

Examples: ROW_COUNT, COLUMN_NAME, URL

1.4. Hungarian Case

The data type as prefix is used to define the variable by developers long ago. This convention is not used anywhere now a day's except local variable declaration.

Example: string m_sName; string strName; int iAge;

1.5. OtherGuideline

1. Use Pascal casing for Class names

```
public class HelloWorld
 {

 }
```

2. Use Pascal casing for Method names

```
public void  PrintMessage(string  name)
 {

 }
```

3. Use Camel casing for variables and method parameters

```
int totalNumber;
   1. public void  PrintMessage(string firstName, string lastName)
{

}
```

4. Use the prefix "I" with Camel Casing for interfaces (Example: **IEntity)**

5. Use Meaningful, descriptive words to name variables. Do not use abbreviations.

   Good:

```
string address;
int salary;
string firstName;
```

   NotGood:

```
string fn;
string strAddress;
int sal;
```

6. Do not use single character variable names like i, n, s etc.

One exception in this case would be variables used for iterations in loops:

```
for (int i = 0; i < count; i++)
{

}
```

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

7. Do not use underscores (_) for local variable names.

8. All member variables must be prefixed with underscore (_) so that they can be identified from other local variables

9. Do not use variable names that resemble keywords.

10. Prefix boolean variables, properties and methods with "is".

Ex: private bool _isFinished; ("_" for member)
private bool isFinished; (local variable)

11. Namespace names should follow the standard pattern

<project name>.<module name>.<folder>

12. Use appropriate prefixes for the UI elements so that you can identify them from the rest.

A brief list is given below. Since .NET has given several controls, you may have to arrive at a complete list of standard prefixes for each of the controls (including third party controls) you are using.

An exception to this is control names. We use Hungarian Notation for these:

Label–	lbl
CheckBox–	chk
LinkLabel–	lnk
CheckedListBox–	chklListbox
ComboBox–	cbo
ProgressBar–	pgb
Control–	ctrl
RadioButton–	rad
Menus–	mnu
RichTextBox–	rtb
Panel–	pnl

Splitter–	spl
TextBox–	txt
PictureBox–	pic
StatusBar–	sba
Datagrid–	dgr
DatagridColumn–	dgrc

ToolBar–	tba
DialogControls–	dlg
ToolTip–	tip
Form–	frm
GroupBox–	gbx
DataTable–	dt
DataSet–	ds
DataRow-	row
Image–	img

13. File name should match with class name.

For example, for the class HelloWorld, the file name should be HelloWorld.cs (or, HelloWorld.vb)

2. Code Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

2. Comments should be in the same level as the code (use the same level of indentation).

Good:

```
// This declares and assigns these variables all in one step.
short levelNumber = 3;
int score = 0;

// This declares a variable, and assigns it at a later point in time.
long aBigNumber;
aBigNumber = -17;
// Here is a byte, which contains any value between 0 and 255
byte aSingleByte = 55;
```

Not Good:

```
// This declares and assigns these variables all in one step.
short levelNumber = 3;
int score = 0;
// This declares a variable, and
//assigns it at a later point in time.
long aBigNumber;
aBigNumber = -17;
// Here is a byte, which contains any value between 0 and 255
byte aSingleByte = 55;
```

3. Curly braces ({}) should be in the same level as the code outside the braces.

Good:

```
private bool SayHello(string name)
{
	DateTime currentTime = DateTime.Now;
	if (string.IsNullOrEmpty(name))
	{
		string message = string.Format("Hello {0}", name);
		Console.WriteLine(message);
		Console.WriteLine(currentTime.ToShortDateString());
	}
	else
	{
		Console.WriteLine("Name is empty");
	}
	return true;
}
```

Bad:

```
private bool SayHello(string name){
DateTime currentTime = DateTime.Now;
	if (string.IsNullOrEmpty(name)){
string message = string.Format("Hello {0}", name);
	Console.WriteLine(message);
	Console.WriteLine(currentTime.ToShortDateString());}
}
```

4. Use one blank line to separate logical groups of code.

Good:

```
private bool SayHello(string name)
{
	DateTime currentTime = DateTime.Now;

	if (string.IsNullOrEmpty(name))
	{
		string message = string.Format("Hello {0}", name);
		Console.WriteLine(message);
		Console.WriteLine(currentTime.ToShortDateString());
	}
	return true;
}
```

Not Good:

```
private bool SayHello(string name)
{
	DateTime currentTime = DateTime.Now;
	if (string.IsNullOrEmpty(name))
	{
		string message = string.Format("Hello {0}", name);
		Console.WriteLine(message);
		Console.WriteLine(currentTime.ToShortDateString());
	}
	return true;
}
```

5. There should be one and only one single blank line between each method inside the class.

6. The curly braces should be on a separate line and not in the same line as if, for etc.

Good:

```
if (isResult == true)
{
	//Do something
}
```

7. Use a single space before and after each operator and brackets.

Good:

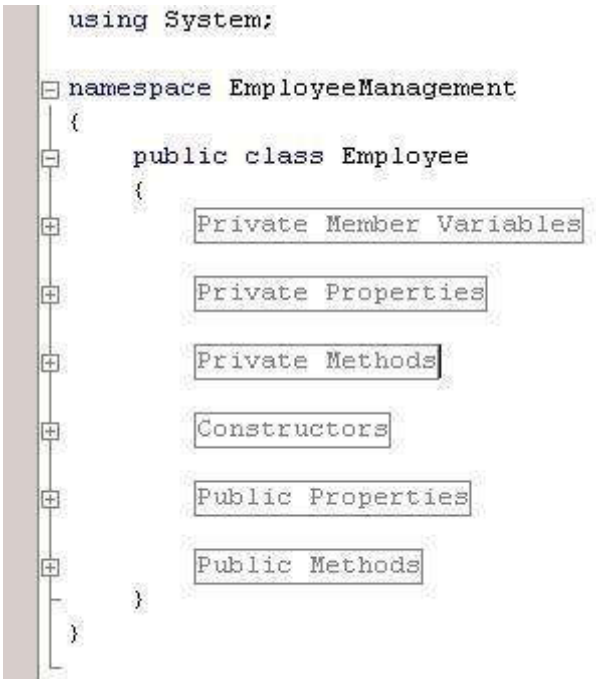
```
if (isResult == true)
{
	for (int i = 0; i < 10; i++)
	{
		//Do something
	}
}
```

Bad:

```
if (isResult==true)
{
	for (int i=0;i<10;i++)
	{
		//Do Something
	}
}
```

8. Use #region to group related pieces of code together. If you use proper grouping using #region, the page should like this when all definitions are collapsed.

Example:



9. Keep private member variables, properties and methods in the top of the file and public members in the bottom.
10. Use default code editor settings provided by Microsoft Visual Studio.
11. Write only one statement and declaration per line.
12. Use parentheses to understand the code written.

3. Good Programming practices

1. Avoid writing very long methods / functions. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

2. Method name should tell what it does. Do not use miss-leading names. If the method name is obvious, there is no need for documentation explaining what the method does.

Good:

```
public void SavePhoneNumber(string phoneNumber)
{
	//Logic to save phone number
}
```

Not Good:

```
public void SaveDetail(string phoneNumber)
{
	//Logic to save phone number
}
```

3. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

```
 if (memberType == MemberType.Registered)
{
	//Do something
}
 else if(memberType==MemberType.Guest)
{
	// Do something if memeber is guest
}

else
{
	//Unexpected Member type throw an exception;
	throw new Exception(string.Format("Unexpected Value :
                                                {0}",memberType.ToString()));
	//If we introduce a new type in future, can easily find problem here
}
```

Not Good:

```
if (memberType == MemberType.Registered)
{
//Do something
} else
{
//Do something if guest is here
//It will create a problem when we introduce a new type
}
```

4. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

   However, using constants is also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

5. Convert strings to lowercase or uppercase before comparing (Applicable only when comparison is not case sensitive). This will ensure the string will match even if the string being compared has a different case.

```
string name = "John";
if (name.ToLower() == "john")
        {
        //Do something
        }
```

   Or use string.Compare function in built

```
string.Compare(name,"john",false);
```

   Here the first and second argument is for comparing strings. The third option is for case ignorance if you pass false then it will ignore case while comparing string and if you put true it will compare string.

6. Use String.Empty instead of“”

```
if(name == string.Empty)
{
   //Do something
}
```

Or use string.IsNullOrEmpty function

```
if(string.IsNullOrEmpty(name))
{
   //Dosomething
}
```

Not Good:

```
if(name=="")
{
   //Do something
}
```

7. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

8. Use Enum whenever required, do not use numbers or string hardcoded.

Good:

```
private enum MailType
{
   HTML,
   PlainText,
   Attachment
}
private void  SendEmail(string  message, MailType  mailType)
{
   switch(mailType)
       {
        case MailType.HTML:
        //Do something
        break;

        case MailType.Attachment:
        //Do something
        break;

        caseMailType.PlainText:
        //Do something
        break;

        default:
        //Do something
        break;
       }

}
```

Not Good:

```
private void  SendEmail(string  message, string mailType)
{
        switch(mailType)
        {
                case "HTML":
                //Do something
                break;

                case "Attachment":
                //Do something
                break;

                case "PlainText":
                //Do something
                break;

                default:
                //Do
                something
                break;
        }
}
```

9. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

10. Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.

11. Never hardcode a path or drive name in code. Get the application path programmatically and use a relative path.

12. Never assume that your code will run from drive "C:". You may never know, some users may run it from the network or from a "Z:".

13. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

14. Every project we should have a generalized error track system (in global.asax), from that we can get appropriate error messages and we can store in database as error log for Do not have more than one class in a single file.

15. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

16. Avoid publi future reference and we can send mail to site administrators also that can help you troubleshoot a problem. We don't have to use try- catch exception handling for each and every function, but for some specific cases like file uploading and sending mail methods we will have to use try- catch exception handling.

17. c methods , it will be safely closed in the finally block.

21. Add Whitesand properties, unless they really need to be accessed from outside the Class. Use "internal" if they are accessed only within the same assembly.

18. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning a List, always return a valid List. If you have no items to return, then return a valid list with 0 items. This will make it easy for the calling application to just check for the "count" rather than doing an additional check for "null".

19. Use the Assembly Info file to fill version number and description.

    If you are opening database connections, sockets, file streams etc., always close them in the finally block. This will ensure that even if an exception occurs after opening the connectionpace around operators, like +, -, ==, etc.

22. Always succeed with the keywords if, else, do, while, for and foreach, with opening and closing parentheses, even though the language does not require it.

23. Avoid using common type systems. Use the language specific aliases

Good :

```
int age;
string firstName;
object addressInfo;
```

Bad :

```
system.Int32 age; String firstName;
Object addressInfo;
```

24. Avoid passing many parameters to function. If you have more than 4-5 parameters use class or structure to pass it.

Good :

```
public void UpdateAddress(Address address)
{
}
```

Bad:

```
public  void UpdateAddress(int addressId, string country, string phoneNumber,
int age , string address1 , string address2 , int pinCode)
{
}
```

25. While working with collection be aware of the below points,

- while returning collections return empty collection instead of returning null when you have no data to return.
- Always check Any() operator instead of checking count i.e. collection.Count >0 and checking of null
- Use foreach instead of for loop while traversing.
- Use Ilist<T>,Ienumerable<T>,Icollection<T> instead of concrete classes e.g. List<>

26. Use object initializers to simplify object creation.

Good :

var employee = new Employee { FirstName = “ABC”, LastName = “PQR”, Manager = “XYZ”};

Bad:

```
var employee = new Employee();
employee.FirstName = “ABC”;
employee.LastName = “PQR”;
employee.Manager = “XYZ”;
```

27. Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in a weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operation.

Consider the following example:

```
public string ComposeMessage(string[] lines)
{
       string message = String.Empty;
       for (int i = 0; i < lines.Length; i++)
       {

            message+=lines[i];
       }
    return message;
}
```

In the above example, it may look like we are just appending to the string object ‘message’. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of the String object.

See the example where the String object is replaced with StringBuilder.

```
public string ComposeMessage(string[] lines)
{
	StringBuilder message = new StringBuilder();
	for (int i = 0; i < lines.Length; i++)
	{
		message.Append(lines[i]);
	}
	return message.ToString();
}
```

4. Architecture

- Always use multi-layer (N-Tier) architecture.

- Never access databases from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.

- Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.

5. ASP.NET

- Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using System.Web.HttpCOntext.Current.Session

- Do not store large objects in session. Storing large objects in session may consume a lot of server memory depending on the number of users. You can use the Cache object for storing large objects which you need frequently. It is much more efficient than this.

- Always use a stylesheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style classes. This will help you to change the UI of your application easily in future.
  Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them

6. ASP.NET Core Web Api

- In Dot Net 5 and before version, We have two classes one for startup and second for Program.cs.
- In Dot Net 6 and later, we don't have a startup class. Startup class merged into Prrogram.cs Class.
- For api use plurals naming and also give api url name as per module name.
- Exa:- Naming Url for Web Api:-
  Bad:
  - For Get all Employees:-
    - domain.com/GetEmployee
    - domain.com/ListEmployee
  - For Save Employees Detail:- Use Post HTTP Verbs and
    - domain.com/SaveEmployee
    - domain.com/CreateEmployee
  - For Get Employees Detail By Id:- Use Get HTTP Verbs and
    - domain.com/GetByID Employee/id

Good:

- For Get all Employees:- Use Get HTTP Verbs
  - domain.com/employees
- For Save Employees Detail:- Use Post HTTP Verbs and
  - domain.com/employees
- For Get Employees Detail By Id:- Use Get HTTP Verbs and
  - domain.com/employees/id

7. Comments

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.
2. Use //or ///for comments. Avoid using/*...*/
3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
4. Do not write comments if the code is easily understandable without comment. The drawback of having a lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.

7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.

8. Perform spelling checks on comments and also make sure proper grammar and punctuation is used.

9. Use xml commenting to describe functions, class and constructor.

8. Exception Handling

1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers use this handy method to ignore no significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.

2. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.

3. Always catch only the specific exception, not generic exception.

Good:

```
void  ReadFromFile(string  fileName)
{
try
    {
       //read from file.
    }

    catch(FileNotFoundException  fileNotFoundException)
    {
       //log error.
       //re-throw exception depending on your case.
       throw;
    }
}
```

Not Good:

```
void  ReadFromFile(string  fileName)
 {
 try
     {
         //read from file.
     }
     catch(Exception  exception)
     {
         //log error.
     }
 }
```

Here we don't know what type of exception is there whether it's a file not found exception or we are having problems with opening file for reading.

4. When you rethrow an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.

Good:

```
try
{
}
Catch (Exception exception)
{
       //Write code for handling exception
       throw;
 }
```

Not Good:

```
try
 {
 }
 catch(Exception  exception)
 {
     //Write code for handling exception
      throw exception;
 }
```

5. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exist in the database, you should try to select the record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. **This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur.**

6. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.

7. If you are using multiple catch then the catch order should be specific to general like following.

```
try
{
}
catch(FileNotFoundException  fileNotFoundException)
{
 //Write code for handling exception
 throw fileNotFoundException;
}
catch(Exception  exception)
{
       //Write code for handling
exception
       throw exception;
}
```

9. Do's for C#

Take care of following.

- Implement IDisposable Interface whenever required.
- Use Switch statement instead of If-else whenever possible.
- Name type name using Noun phrases or adjective phrases.
- Use Constant and readonly whenever required.
- Finish every if-elseif statement with the else part.
- Always write default in the switch statement.
- Avoid nested loops as much as possible.
- Use Inexplicit comparison i.e. use if(IsValid) instead of if(IsValid==true).
- Declare and initialize variables as late as possible.
- Use using statements as much as possible.
- Always use resource files for the text of labels, buttons and other controls.

10. Don'ts for C#:

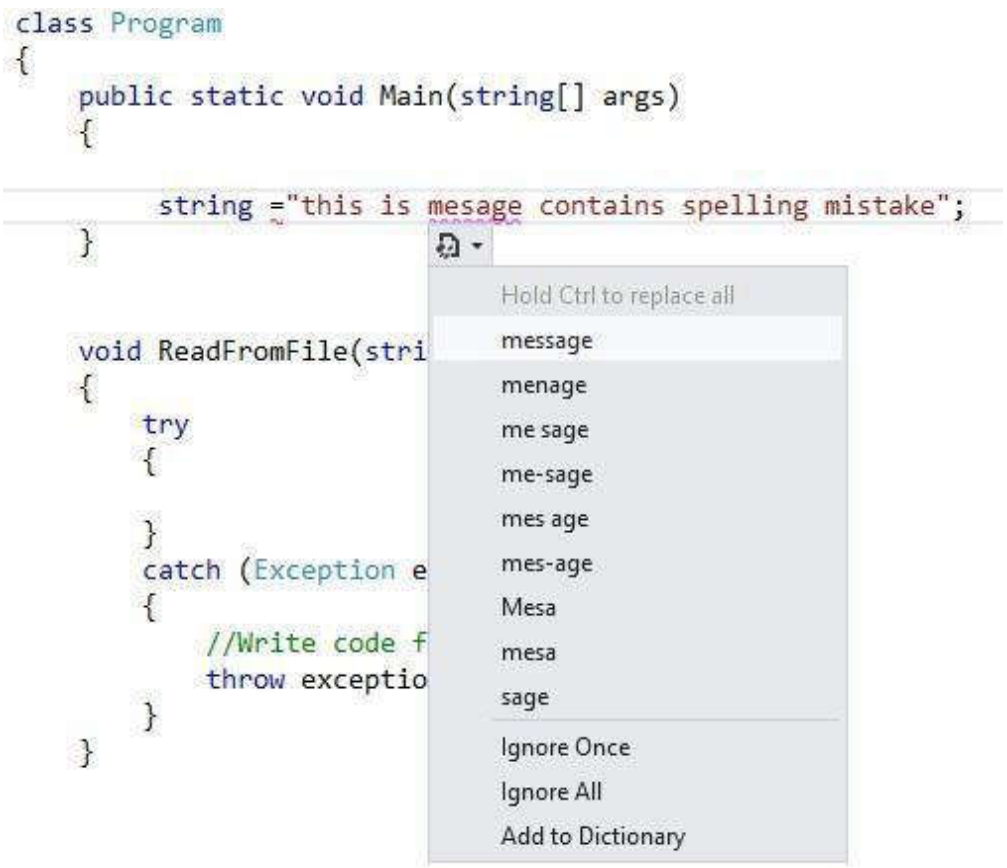
- Avoid using static classes and static variables.
- Don't hide inherited members with new keywords.
- Don't write methods with hundreds of lines. As a thumb rule, if methods contain more than 25 lines of code then it should be subdivided into other methods.
- Do not pass parameters of methods as null.
- Properties, Methods and Arguments representing string should never be NULL.
- Do not put multiple classes in a single file.
- Files/Class should not have 1000 lines of code; it's better to have that class divided into multiple classes.
- No line should exceed 200 hundred characters.
- Do not make spelling mistakes in code; it leaves a bad impression.

11. Spell checking in code files, html files

- There is a very good spell check available for Visual Studio which takes care of spelling mistakes made by developers. You can download that from the following link.

  http://visualstudiogallery.msdn.microsoft.com/a23de100-31a1-405c-b4b7-d6be40c3dfff

- It helps correct spelling mistakes from the code and if there is a mistake it will highlight as underline.



12. Code quality

Based on requirement and with suggestion of customer developer has to use below tools to maintain the code quality.

12.1. Code Analyses

The developer should use Code Analyses to check their code. The Microsoft Managed Recommended Rules should be used and all code except auto-generated code should comply with these rule.

12.2. Style Cop

The developer should use the latest Style Cop available from https://stylecop.codeplex.com/)and should make sure that all code complies with the supplied rule set. Auto generated code are exempt from this rule.

12.3. Re-Sharper

Visual studio plugin which helps developers with a great deal of frequent software development and maintenance tasks, such as finding unused code, complying with naming guidelines, detecting possible runtime exceptions.

12.4. Exceptions to the rules

- In some cases it can be that the coding rules can't be followed. The developer is not allowed to disable those rules by themselves.
  A developer can suggest an exception by putting following comment:
  // QualityCheck: Reason why you want the exception.

- Avoid writing very long methods / functions. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

- Method name should tell what it does. Do not use miss-leading names. If the method name is obvious, there is no need for documentation explaining what the method does.

  Example:

```
public void SavePhoneNumber(string phoneNumber)
{
```

```
            //Logic to save phonenumber
        }
```

- Do not hard code numbers. Use constants instead. Declare constant in the top of the file and use it in your code. However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

- Convert strings to lowercase or uppercase before comparing(Applicable only when comparison is not case sensitive). This will ensure the string will match even if the string being compared has a different case.

```
String name ="John";
if(name.ToLower() =="john")
 {
              //Do something
 }
```

Or use string.Compare functioninbuilt

```
string.Compare(name,"john",false);
```

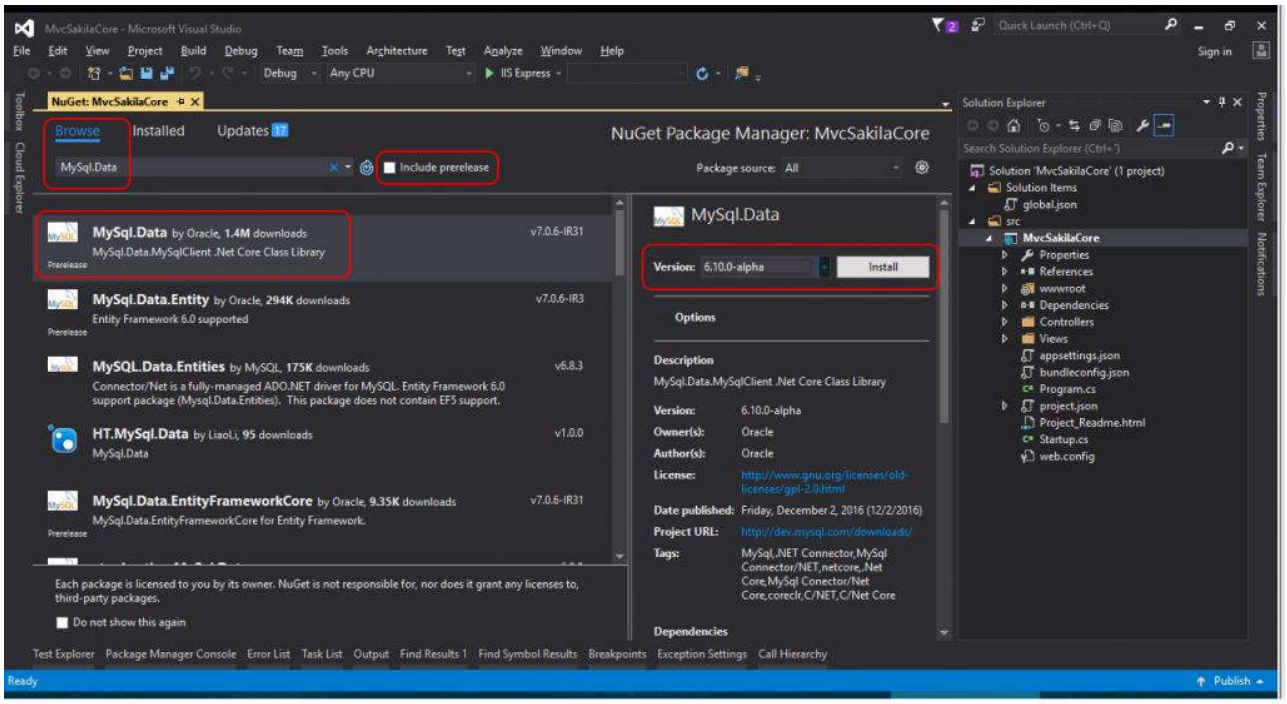
Here first and second argument is for comparing string third option is for case ignorance if you pass false then it will ignore case while comparing string and if you put true it will compare string.

- Use String.Empty instead of“”
- Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
- Never hardcode a path or drive name in code. Get the application path programmatically and use a relative path.
- Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
- Every project we should have a generalized error track system (in global.asax), from that we can get appropriate error messages and we can store them in the database as error log for future reference and we can send mail to site administrators also that can help you troubleshoot a problem. We don’t have to use try-catch exception handling for each and every function, but for some specific case like file uploading and sending mail methods we will have to use try- catch exception handling.
- Use the Assembly Info file to fill version number and description.
- If you are opening database connections, sockets, file streams etc., always close them in the finally block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the finally block.
- Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in a weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

13. ASP.Net Core Best practices

13.1 Initialize connection with MySql and MS SQL Server Database:-

- **Initialize connection with MySql :-**
  - Installing MySQL Connector/NET Core package:-
    - In order to use MySQL Connector/NET it's required to add its nuget package.
    - In Solution Explorer, right-click On Project > Manage Nu Get Packages.
    - In the NuGet dialog, Browse "My Sql .Data" package using version 6.10.0 or above:



    - Adding the connection string:-
      - Add your connection string in the appsettings.json file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "server=localhost;port=3306;database=sakila;user=test;password=test"
  },
  "Logging": {
    "IncludeScopes": false,
```

- Adding data model class:-

  - For this example a Film class will be used. It contains the database fields as properties we want to show in our application.

  - Add a new class named "Film" inside Models folder:

  - **Film.cs:-**

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using MySql.Data.MySqlClient;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MvcSakilaCore.Models
{
  public class Film
  {
    private SakilaContext context;

    public int FilmId { get; set; }

    public string Title { get; set; }

    public string Description { get; set; }

    public int ReleaseYear { get; set; }

    public int Length { get; set; }

    public string Rating { get; set; }
  }
}
```

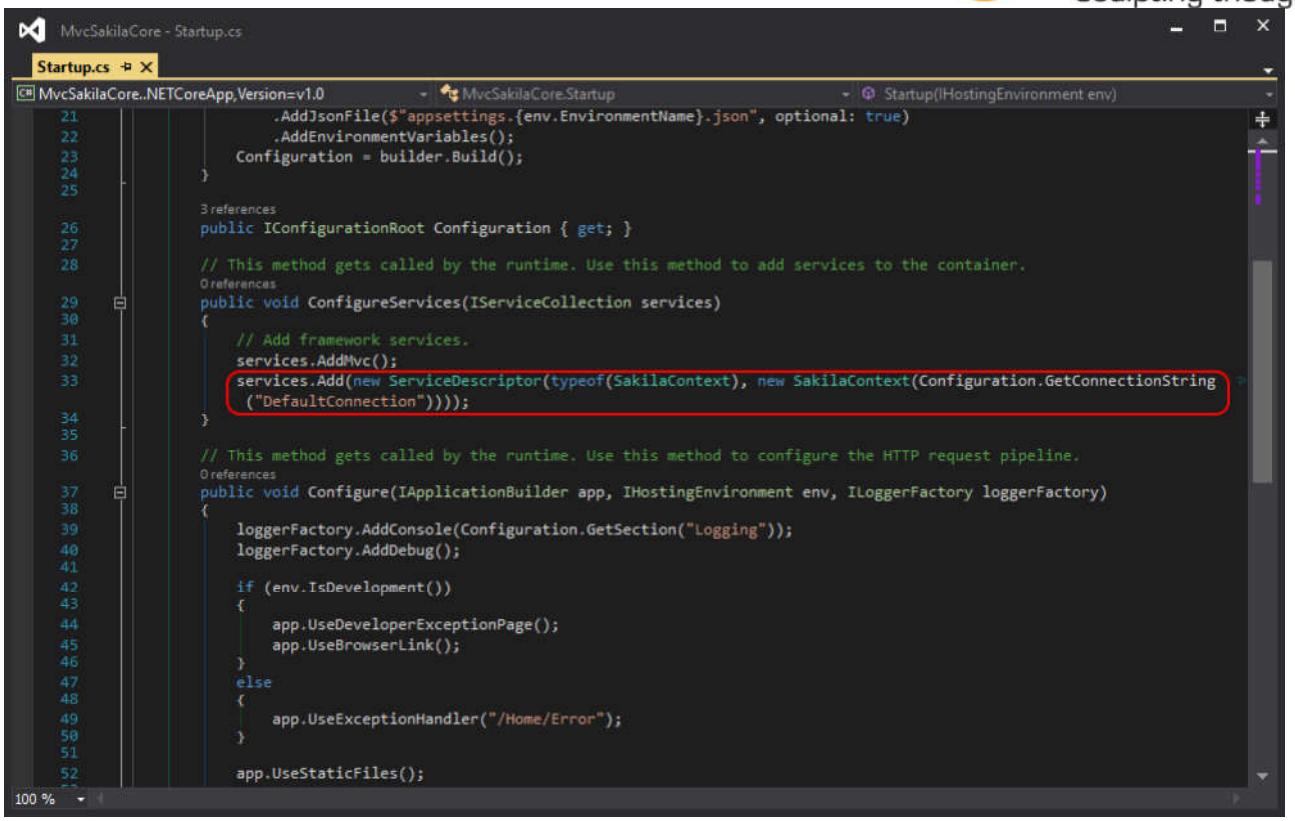
- Create a new SakilaContext class that will contains the connections and Sakila database entities:

- **SakilaContext.cs:-**

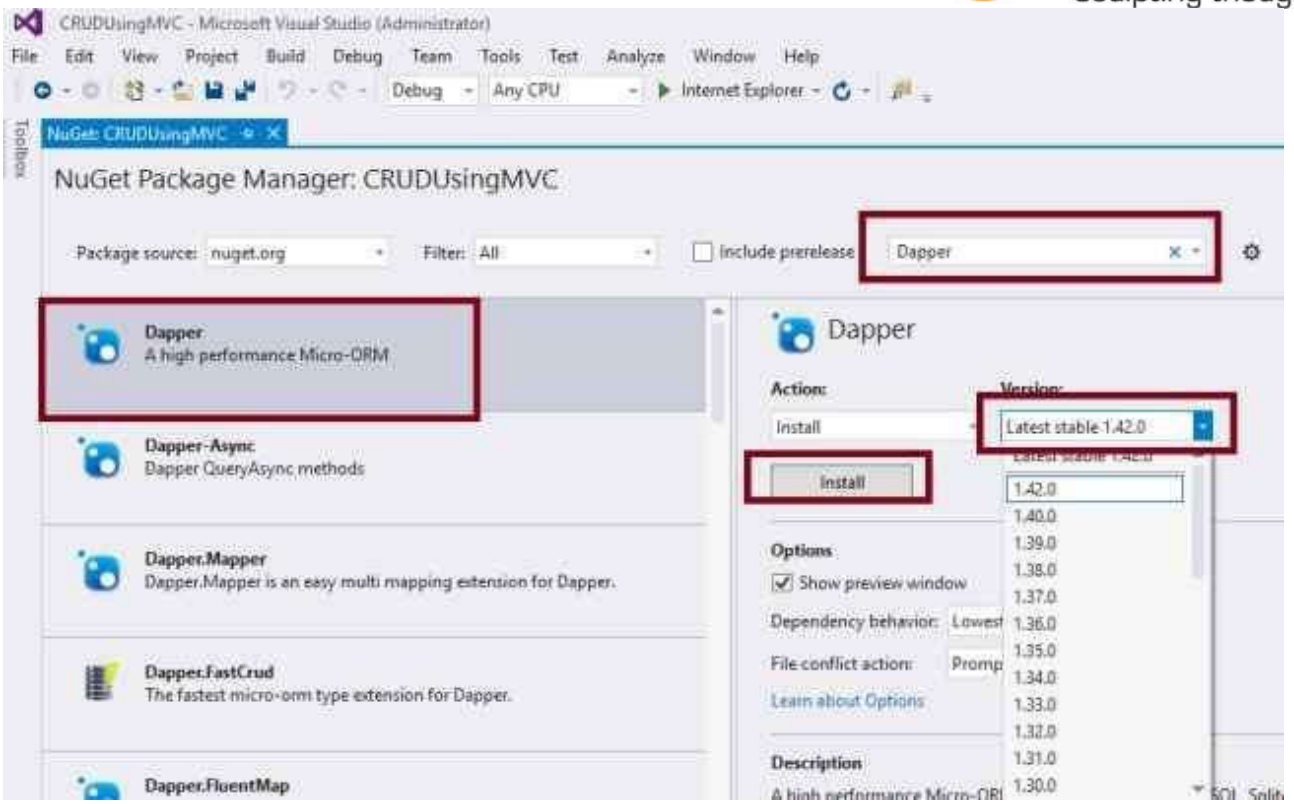
```
using MySql.Data.MySqlClient;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;

namespace MvcSakilaCore.Models
{
  public class SakilaContext
  {
    public string ConnectionString { get; set; }

    public SakilaContext(string connectionString)
    {
      this.ConnectionString = connectionString;
    }

    private MySqlConnection GetConnection()
    {
      return new MySqlConnection(ConnectionString);
    }

    public List<Film> GetAllFilms()
    {
      List<Film> list = new List<Film>();

      using (MySqlConnection conn = GetConnection())
      {
        conn.Open();
        MySqlCommand cmd = new MySqlCommand("SELECT *
FROM film", conn);
        using (MySqlDataReader reader = cmd.ExecuteReader())
        {
          while (reader.Read())
          {
```

```
                list.Add(new Film()
                {
                  FilmId = reader.GetInt32("film_id"),
                  Title = reader.GetString("title"),
                  Description = reader.GetString("description"),
                  ReleaseYear = reader.GetInt32("release_year"),
                  Length = reader.GetInt32("length"),
                  Rating = reader.GetString("rating")
                });
              }
            }
          }

          return list;
        }
      }
    }
```

- In order to be able to use our SakilaContext it's required to register the instance as a service in our application. To do this add the code line in the Startup.cs file:



- **Initialize connection with Sql Using Dapper Service :-**
    - Add The Reference of Dapper ORM into Project.
        - Right click on Solution ,find Manage NuGet Package manager and click on it.
        - After as shown into the image and type in search box "*dapper*".
        - Select Dapper as shown into the image .
        - Choose version of dapper library and click on install button.


CRUDUsingMVC - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
Any CPU
Internet Explorer
NuGet: CRUDUsingMVC
NuGet Package Manager: CRUDUsingMVC
Package source: nuget.org
Filter: All
Include prerelease
Dapper
Dapper
A high performance Micro-ORM
Dapper-Async
Dapper QueryAsync methods
Dapper.Mapper
Dapper.Mapper is an easy multi mapping extension for Dapper.
Dapper.FastCrud
The fastest micro-orm type extension for Dapper.
Dapper.FluentMap
Dapper
Action:
Install
Version:
Latest stable 1.42.0
Install
1.42.0
1.40.0
1.39.0
1.38.0
1.37.0
1.36.0
1.35.0
1.34.0
1.33.0
1.32.0
1.31.0
1.30.0
Options
Show preview window
Dependency behavior:
File conflict action:
Learn about Options
Description

- Create Model Class.

```
public class ComplaintModel
 {
        [Display(Name = "Complaint Type")]
    [Required]
    public string ComplaintType { get; set; }
    [Display(Name = "Complaint Description")]
    [Required]
    public string ComplaintDesc { get; set; }

}
```

- Create Table and Stored procedure.

- Create Repository class For database connection.

➢ ComplaintRepo.cs:

```
public class ComplaintRepo
 {
    SqlConnection con;
    //To Handle connection related activities
    private void connection()
    {
        string constr =
ConfigurationManager.ConnectionStrings["SqlConn"].ToString();
        con = new SqlConnection(constr);
    }
    //To Add Complaint details
    public string AddComplaint(ComplaintModel Obj)
    {
        DynamicParameters ObjParm = new DynamicParameters();
        ObjParm.Add("@ComplaintType", Obj.ComplaintType);
        ObjParm.Add("@ComplaintDesc", Obj.ComplaintDesc);
        ObjParm.Add("@ComplaintId",
dbType:DbType.String,direction:ParameterDirection.Output,size:521558
5);
        connection();
        con.Open();
```

```
con.Execute("AddComplaint",ObjParm,commandType:CommandType.
StoredProcedure);
        //Getting the out parameter value of stored procedure
        var ComplaintId = ObjParm.Get<string>("@ComplaintId");
        con.Close();
        return ComplaintId;

    }
 }
```

13.2 Bootstrap Validation :-

- You can use different validation classes to provide valuable feedback to users.
- Add either .was-validated or .needs-validation to the <form> element, depending on whether you want to provide validation feedback before or after submitting the form.
- The input fields will have a green (valid) or red (invalid) border to indicate what's missing in the form.
- **Example :-**
  Following screenshot of an ASP.NET MVC Core application without bootstrap styles.

Name

The Name field is required.

Email

The Email field is required.

Password

The Password field is required.

ConfirmPassword

The ConfirmPassword field is required.

Url

Submit

- **Razor Code of above page:-**

```
@model PersonViewModel
@{
   ViewData["Title"] = "Register";
}

<form asp-action="Register" method="POST" class="needs-validation"
     novalidate>
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
    <span asp-validation-for="Name"></span>
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input class="form-control" asp-for="Email" />
    <span asp-validation-for="Email"></span>
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input class="form-control" asp-for="Password" />
    <span asp-validation-for="Password"></span>
  </div>
  <div class="form-group">
    <label asp-for="ConfirmPassword"></label>
    <input class="form-control" asp-for="ConfirmPassword" />
    <span asp-validation-for="ConfirmPassword"></span>
  </div>
  <div class="form-group">
    <label asp-for="Url"></label>
    <input class="form-control" asp-for="Url" />
    <span asp-validation-for="Url"></span>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

```
@section scripts
{
   <partial name="_ValidationScriptsPartial" />
}
```

- **Model Class:-**

```
public class PersonViewModel
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required, DataType(DataType.EmailAddress), EmailAddress]
    public string Email { get; set; }
    [Required, DataType(DataType.Password),
Compare(nameof(ConfirmPassword))]
    public string Password { get; set; }
    [Required, DataType(DataType.Password)]
    public string ConfirmPassword { get; set; }
    [Url]
    public string Url { get; set; }
}
```

- We can customize the colour and font size of the error message, for that you need to create a style class - field-validation-error. ASP.NET MVC Core used to add this class for the SPAN element. So if we add an style class like this, it will display the error message red color.

```
.field-validation-error {
  color:red;
  font-size: smaller;
}
```

- Now if we run the application, it will display the error message in red colour with smaller font. Bootstrap supports styling validation controls with the help of server side code implementation. From Bootstrap documentation.

- We recommend using client-side validation, but in case you require server-side validation, you can indicate invalid and valid form fields with .is-invalid and .is-valid. Note that .invalid-feedback is also supported with these classes.

- Since ASP.NET Core MVC is using JQuery validation with unobtrusive script for data annotations validation, we can customize the validator object and configure the validClass and errorClass properties, like this.

```
var settings = {
    validClass: "is-valid",
    errorClass: "is-invalid"
};
$.validator.setDefaults(settings);
$.validator.unobtrusive.options = settings;
```

- You can place the above code snippet in the _ValidationScriptsPartial.cshtml file. And now if you run your application, you will be able to see the Bootstrap validation style messages.

13.3 Asynchronous programming:-

- Asynchronous programming is very popular with the help of the async and await keywords in C#. When we are dealing with UI, and on button click, we use a long-running method like reading a large file or something else which will take a long time, in that case, the entire application must wait to complete the whole task. In other words, if any process is blocked in a synchronous application, the whole application gets blocked, and our application stops responding until the whole task completes.

- 'Asynchronous programming is very helpful in this condition. By using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the entire task.

- We will get all the benefits of traditional Asynchronous programming with much less effort with the help of async and await keywords.

- Suppose we are using two methods as Method1 and Method2 respectively, and both the methods are not dependent on each other, and Method1 takes a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for the completion of this method, and then it will execute Method2. Thus, it will be a time-intensive process even though both methods are not depending on each other.

- We can run all the methods parallelly by using simple thread programming, but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming, but if we use the async and await keywords, we will get the solutions in much less code.

- Also, we are going to see more examples, and if any third Method, as Method3 has a dependency of method1, then it will wait for the completion of Method1 with the help of await keyword.

- Async and await in C# are the code markers, which marks code positions from where the control should resume after a task completes.

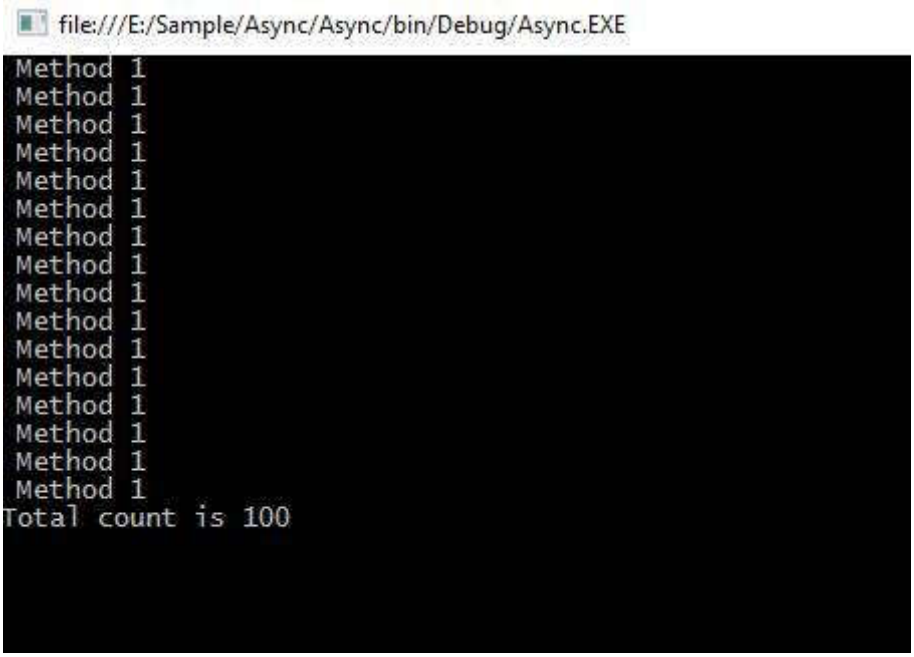
- **Example 1:-**

```
class Program
{
    static async Task Main(string[] args)
    {
        await callMethod();
        Console.ReadKey();
    }

    public static async Task callMethod()
    {
        Method2();
        var count = await Method1();
        Method3(count);
    }
```

```
    public static async Task<int> Method1()
    {
        int count = 0;
        await Task.Run(() =>
        {
            for (int i = 0; i < 100; i++)
            {
                Console.WriteLine(" Method 1");
                count += 1;
            }
        });
        return count;
    }

    public static void Method2()
    {
        for (int i = 0; i < 25; i++)
        {
            Console.WriteLine(" Method 2");
        }
    }

    public static void Method3(int count)
    {
        Console.WriteLine("Total count is " + count);
    }
}
```

- In the code given above, Method3 requires one parameter, which is the return type of Method1. Here, await keyword is playing a vital role for waiting for Method1 task completion.

- Output:-



13.4 Caching:-

- Caching refers to the process of storing frequently used data so that those data can be served much faster for any future requests. So we take the most frequently used data and copy it into temporary storage so that it can be accessed much faster in future calls from the client. If we try to explain with a simple example, Let User-1 request some data and it takes 12-15 seconds for the server to fetch the data. While fetching, we will make a copy of our fetched data parallelly to any temporary storage. So now, when User-2 requests the same data, this time we will simply serve his from the cache and it will take only 1-2 seconds for the response as we already stored the response in our cache.

- There are two important terms used with cache, **cache hit** and **cache miss**. A cache hit occurs when data can be found in a cache and a cache miss occurs when data can't be found in the cache.

- Caching significantly improves the performance of an application, reducing the complexity to generate content. It is important to design an application so that it never depends directly on the cached memory. The application should only cache data that doesn't change frequently and use the cache data only if it is available.

- ASP.NET Core has many caching features. But among them the **two** main types are,
  1. In-memory caching
  2. Distributed Caching

- **In-memory caching :-**

  ✓ An **in-memory** cache is stored in the memory of a single server hosting the application. Basically, the data is cached within the application. This is the easiest way to drastically improve application performance.

  ✓ The main advantage of In-memory caching is it is much quicker than distributed caching because it avoids communicating over a network and it's suitable for small-scale applications. And the main disadvantage is maintaining the consistency of caches while deployed in the cloud.

  ✓ **Example:-** First create an ASP.NET Core web API application.

  ✓ Now inside the Startup.cs file just add the following line. This will add a non-distributed in-memory caching implementation to our application.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMemoryCache();
   //Rest of the code
}
```

✓ Now let's create a new controller "EmployeeController". And in this controller we will implement our cache.

```
[Route("api/[controller]")]
[ApiController]
public class EmployeeController : ControllerBase
{
    private readonly IMemoryCache _memoryCache;
    private readonly ApplicationContext _context;
        public EmployeeController(IMemoryCache memoryCache,
    ApplicationContext context)
    {
        _memoryCache = memoryCache;
        _context = context;
    }

    [HttpGet]
    public async Task<IActionResult> GetAllEmployee()
    {
        var cacheKey = "employeeList";
        //checks if cache entries exists
                            if (!_memoryCache.TryGetValue(cacheKey, out
                    List<Employee> employeeList))
        {
            //calling the server
            employeeList = await _context.Employees.ToListAsync();

            //setting up cache options
            var cacheExpiryOptions = new MemoryCacheEntryOptions
            {
                AbsoluteExpiration = DateTime.Now.AddSeconds(50),
                Priority = CacheItemPriority.High,
                SlidingExpiration = TimeSpan.FromSeconds(20)
            };
            //setting cache entries
                                    _memoryCache.Set(cacheKey, employeeList,
                    cacheExpiryOptions);
        }
        return Ok(employeeList);
    }
      }
```

- ✓ This is a pretty simple implementation. We are simply checking if any cached value is available for the specific cache key. If it exists it will serve the data from the cache, if not we will call our service and save the data in the cache.

- **Distributed Caching:-**

  - ✓ Distributed cache is a cache that can be shared by one or more applications and it is maintained as an external service that is accessible to all servers. So the distributed cache is external to the application.

  - ✓ The main advantage of distributed caching is that data is consistent throughout multiple servers as the server is external to the application, any failure of any application will not affect the cache server.

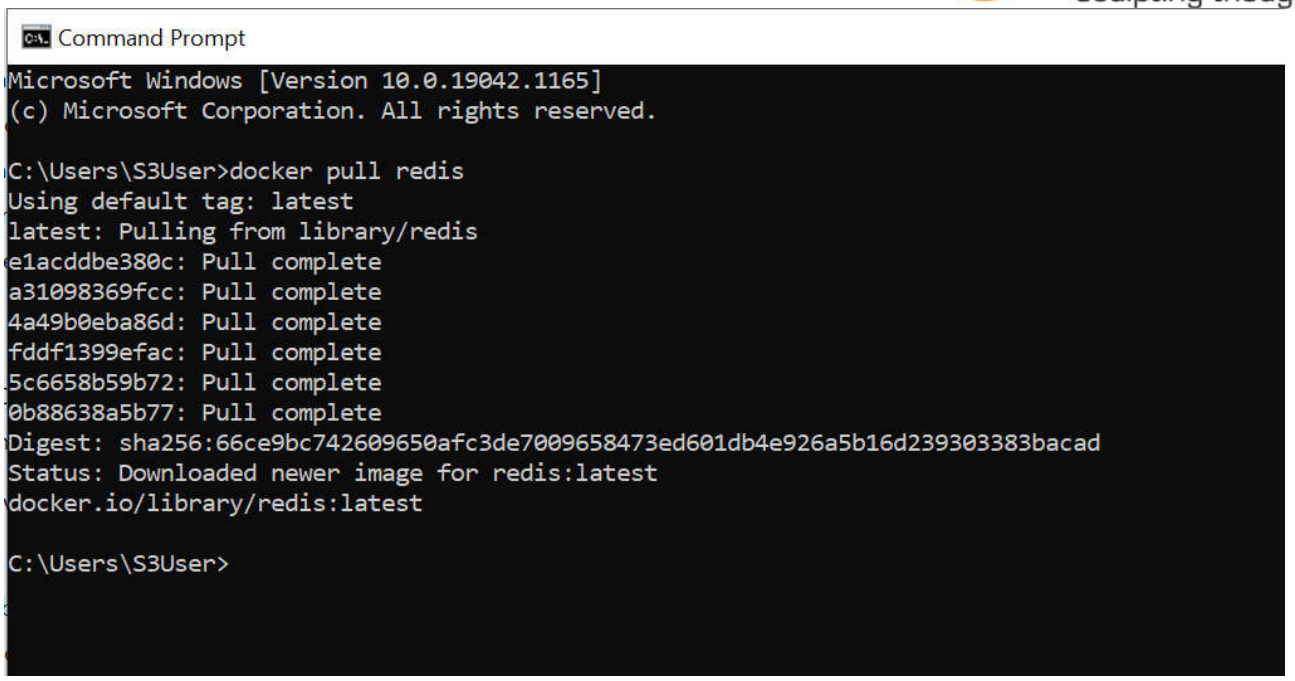
  - ✓ Here we will try to implement Distributed Caching with Redis.

  - ✓ **Redis** is an open-source(BSD licensed), in-memory data structure store, used as a database cache and message broker. It is really fast key-value based database and even NoSQL database as well. So Redis is a great option for implementing highly available cache.

  - ✓ Setting up Raids Cache

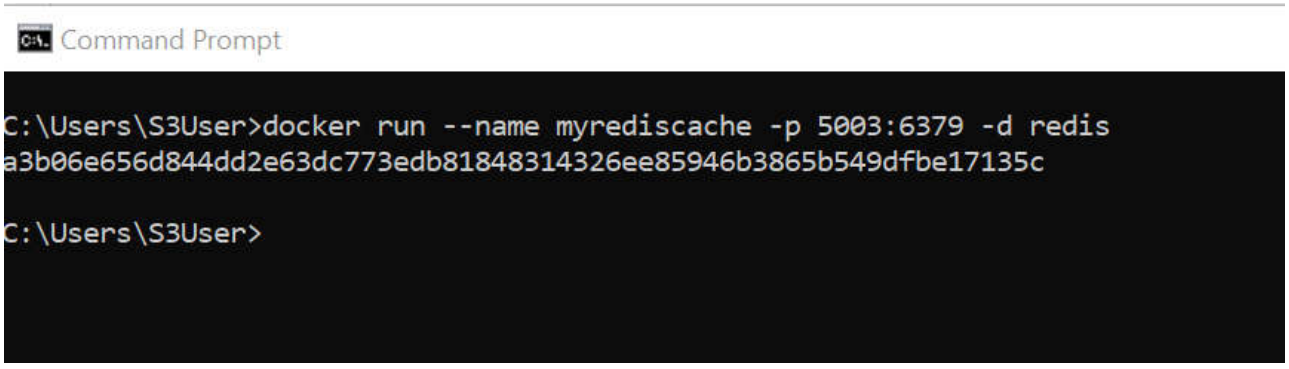
    - ➢ Pull docker Redis image from docker hub.

```
docker pull redis
```



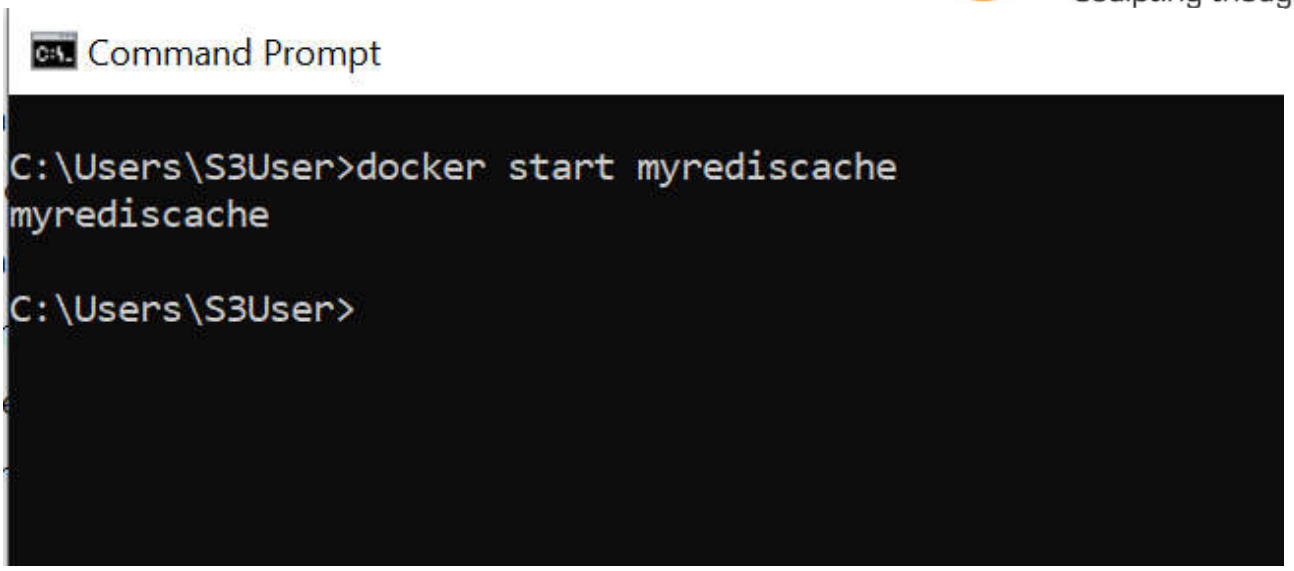
- Run redis images by mapping Redis port to our local system port.

```
docker run --name myrediscache -p 5003:379 -d redis
```



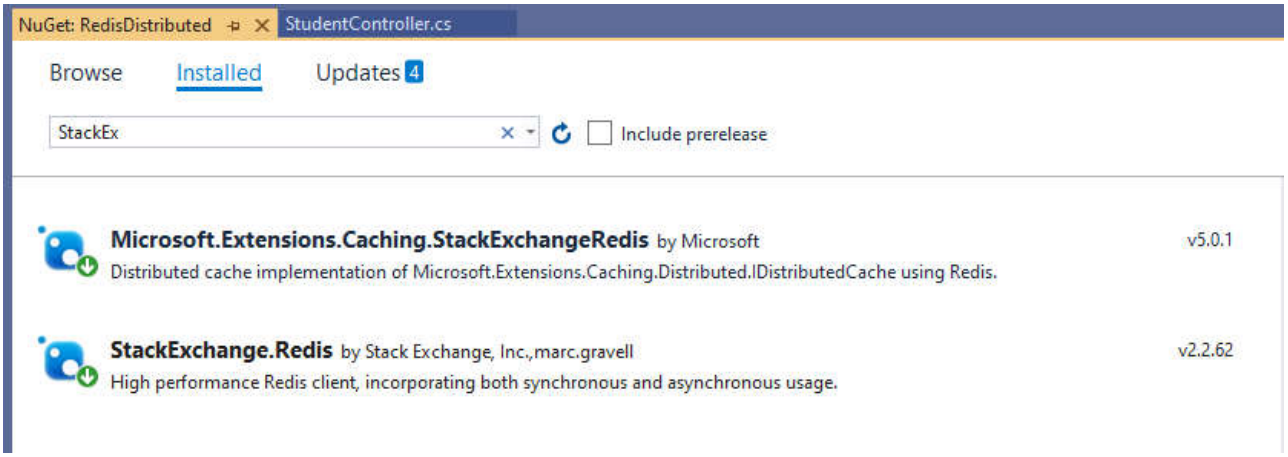
- Start the container.

```
docker start myrediscache
```



- As our Redis is set up now let's go for the implementation of **Distributed caching with ASP.NET Core** Application.

  - Create an ASP.NET Core Web API project and install the following library using Nuget Package Manager.



- As we have already added our required package, now register the services in **Startup.cs** file.

```
public void ConfigureServices(IServiceCollection services)
{
   //Rest of the code
   services.AddStackExchangeRedisCache(options => {
                    options.Configuration =
           Configuration.GetConnectionString("Redis");
      options.InstanceName = "localRedis_";
   });
}
```

- Here, we set "options.InstanceName" it will just act as a **prefix** to our key name on the redis server. Ex. if we set a cache name employeelist in the redis server it will be something like localRedis_employeelist.

- And we will provide the configuration-related settings for the Redis in appsettings.json.

```
{
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "Redis": "localhost:5003",
    "DefaultConnection": "Data Source=.;Initial
Catalog=BuildingDataDB;Integrated Security=True"
  }
}
```

- Create a helper class "**DistributedCacheExtensions**" where we will Get and Set Values from and to Redis Cache.

```
public static class DistributedCacheExtension {
    public static async Task SetRecordAsync <T> (this
IDistributedCache cache, string recodeId, T data, TimeSpan?
absoluteExpireTime = null, TimeSpan ? slidingExpirationTime = null)
{
        var options = new DistributedCacheEntryOptions();
        options.AbsoluteExpirationRelativeToNow =
        absoluteExpireTime ?? TimeSpan.FromSeconds(60);
        options.SlidingExpiration = slidingExpirationTime;
        var jsonData = JsonSerializer.Serialize(data);
        await cache.SetStringAsync(recodeId, jsonData, options);
        }
         public static async Task <T> GetRecordAsync <T > (this
        IDistributedCache cache, string recordId) {
            var jsonData = await cache.GetStringAsync(recordId);
            if (jsonData is null) {
                return default (T);
            }
            return JsonSerializer.Deserialize < T > (jsonData);
    }
}
```

- Here this code is pretty self-explanatory. In the "**SetRecodeAsync**" method we are saving the data to the Redis Cache. Here we have configured the IDistributedCache server
with **AbsoluteExpirationRelativeToNow** and **SlidingExpiration(Line 12 & Line 13)** and we have already discussed these terms in our In-memory Caching section.

- And in the "**GetRecordAsync**" we are getting the cached value depending on some recodeKey.

- Now we will create a controller named "StudentController",

```
public class StudentController : ControllerBase
{
        private readonly ApplicationContext _context = null;
        private readonly IDistributedCache _cache;

        public StudentController(ApplicationContext context, IDistributedCache
cache)
        {
            _context = context;
            _cache = cache;
        }

        [HttpGet]
        public async Task<ActionResult<List<Student>>> Get()
        {
            var cacheKey = "GET_ALL_STUDENTS";
            List<Student> students = new List<Student>();
                  var data = await
                _cache.GetRecordAsync<List<Student>>(cacheKey);
            if (data is null)
            {
                Thread.Sleep(10000);
                data = _context.Student.ToList();
                await _cache.SetRecordAsync(cacheKey, data);
            }
            return data;
}
```

13.5 View Component:-

- ViewComponent was introduced in ASP.NET Core MVC. It can do everything that a partial view can and can do even more.
- ViewComponents are completely self-contained objects that consistently render html from a razor view.
- ViewComponents are very powerful UI building blocks of the areas of application which are not directly accessible for controller actions.
- Let's suppose we have a page of social icons and we display icons dynamically. We have separate settings for color, urls and names of social icons and we have to render icons dynamically
- **Example:-**

```
public class SocialLinksViewComponent : ViewComponent
      {
         List<SocialIcon> socialIcons = new List<SocialIcon>();
         public SocialLinksViewComponent()
         {
            socialIcons = SocialIcon.AppSocialIcons();
         }

         public async Task<IViewComponentResult> InvokeAsync()
         {
            var model = socialIcons;
                  return await
          Task.FromResult((IViewComponentResult)View("SocialLinks", model));
         }

  }
```

13.6 Dynamic Bootstrap Dialog:-

- The **Modal** is a popup window or dialog box that displayed over the current web page. It is very useful to display HTML content/elements on a single page. If your web application uses Bootstrap, the modal popup can be easily implemented on the web pages. Bootstrap's modal plugin helps to add a dialog window to the website site for lightboxes, popup elements, or custom content.

- Creating a modal popup is very easy with the **Bootstrap modal** component. So, if your web application already uses Bootstrap, it's always a good idea to use Bootstrap for populating modal dialog. Because it does not require any third-party jQuery plugin. Not only the static content but also you can load external URL or **dynamic content in a modal popup with Bootstrap**.

- In this tutorial, we will show how you can load content from an external URL in **Bootstrap modal popup**. Also, you will know how to load dynamic content from another page via jQuery Ajax and display it in Bootstrap modal popup. Using our example script, you can pass data, variables, or parameters to the external URL and get dynamic content via jQuery Ajax.

- Before using the Bootstrap to create a modal popup, include the Bootstrap and jQuery library first.

```
<!-- jQuery library -->
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>


<!-- Bootstrap library -->
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script
>
```

- The following HTML contains a button and a modal dialog. This button (`.openBtn`) triggers the Bootstrap modal for showing the content from another file.

```
<!-- Trigger the modal with a button -->
<button type="button" class="btn btn-success openBtn">Open Modal</button>


<!-- Modal -->
<div class="modal fade" id="myModal" role="dialog">
    <div class="modal-dialog">
        <!-- Modal content-->
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-
dismiss="modal">&times;</button>
                <h4 class="modal-title">Modal with Dynamic Content</h4>
            </div>
            <div class="modal-body">


            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-default" data-
dismiss="modal">Close</button>
            </div>
        </div>
    </div>
</div>
```

➢ Load Content from Another Page in Bootstrap Modal

- This example shows how to load the content from an external URL in the Bootstrap modal popup.
- **JavaScript Code:**
  By clicking the Open Modal (.openBtn) button, the content is loaded from another page (content.html) and shows on the modal popup (#myModal).

```
<script>
$('.openBtn').on('click',function(){
    $('.modal-body').load('content.html',function(){
        $('#myModal').modal({show:true});
    });
});
</script>
```

➢ Load Dynamic Content from Database in Bootstrap Modal

This example shows how to load the dynamic content based on parameter pass into the external URL using PHP and MySQL.

JavaScript Code:
By clicking the Open Modal (.openBtn) button, the dynamic content is loaded from another PHP file (getContent.php) based on the ID and shows on the modal popup (#myModal).

```
<script>
$('.openBtn').on('click',function(){
    $('.modal-body').load('getContent.php?id=2',function(){
        $('#myModal').modal({show:true});
    });
});
</script>
```

13.7 Generic Repository Service:-

- **Repository pattern** use to separate the logic that retrieves the data and maps it to an entity model from the business logic that acts on the model. This enables the business logic to be agnostic to the type of data that comprises the data source layer.

- A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection.

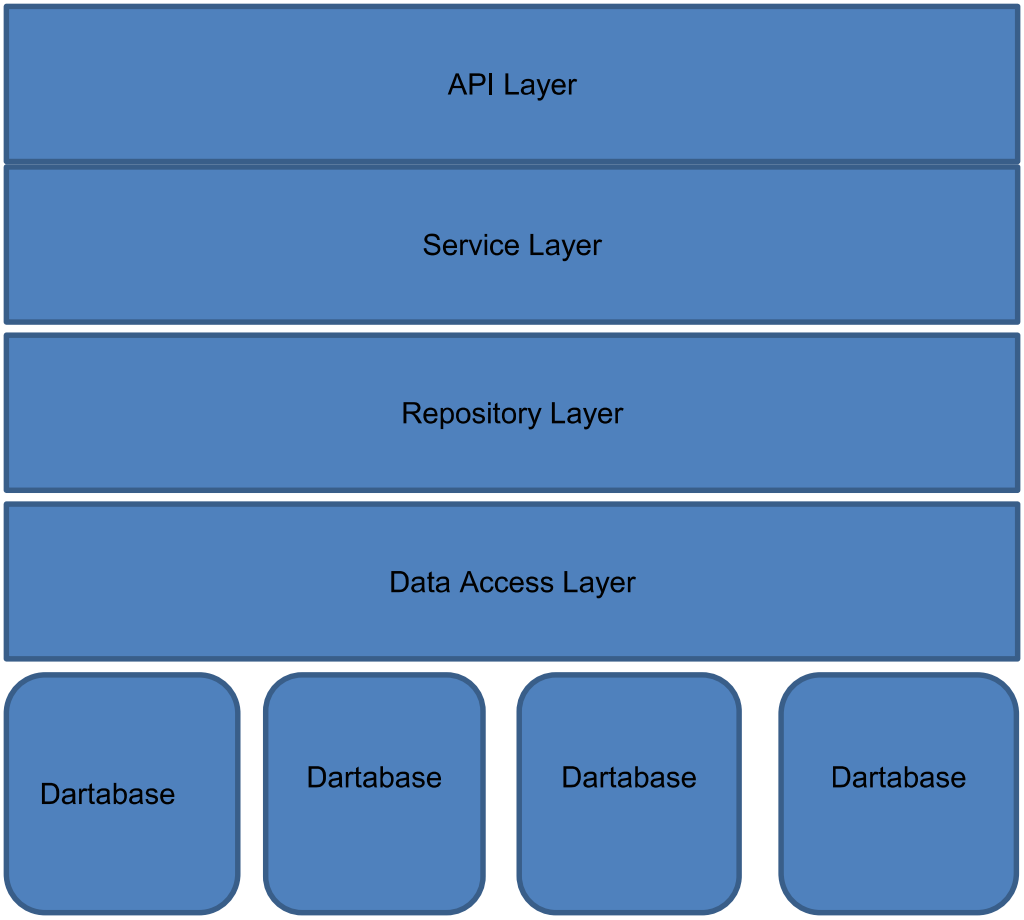
- Benefits of the Repository Pattern

    - Centralization of the data access logic.
    - Substitution point for the unit tests.
    - Flexible architecture that can be adapted as the overall design of the application evolves.

- In order to enable your application to meet all the *illities:-*
    - Scalability
    - Adaptability
    - Testability
    - Re-usability

- It's important to break up the development of your application into layers. Each layer can then be injected.

- This provides levels of abstractions for your various layers in that they do not necessarily explicitly care where the data from each layer is persisted and retrieved from only that it conforms to an explicit data contract.



- This also enables ease of testing by providing the ability to inject Mock or Fake abstracted classes to provide data.

- **Unit Of Work :-**

  - A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

- **Example:**

```
public interface IRepository<T> : IDisposable where T : class
 {
          IQueryable<T> Query(string sql, params object[] parameters);

         T Search(params object[] keyValues);

         T Single(Expression<Func<T, bool>> predicate = null,
          Func<IQueryable<T>, IOrderedQueryable<T>> orderBy = null,
          Func<IQueryable<T>, IIncludableQueryable<T, object>> include = null,
          bool disableTracking = true);

           void Add(T entity);
           void Add(params T[] entities);
           void Add(IEnumerable<T> entities);


           void Delete(T entity);
           void Delete(object id);
           void Delete(params T[] entities);
           void Delete(IEnumerable<T> entities);


           void Update(T entity);
           void Update(params T[] entities);
           void Update(IEnumerable<T> entities);
    }
```

13.8 Auto Register interface (Scrutor):-

- ASP.NET Core uses dependency injection throughout the core of the framework. Consequently, the framework includes a simple DI container that provides the minimum capabilities required by the framework itself.
- There are also many third-party .NET DI libraries that provide many more capabilities and features. I wrote quite a while back about using StructureMap in ASP.NET Core (though if you're using StructureMap, you should probably take a look at Lamar instead), but there are many containers to choose from, for example:
    - AutoFac
    - Windsor
    - StructureMap/Lamar
    - Simple Injector
    - Ninject
    - Dryloc

- Scrutor is not a new DI container. Under the hood it uses the built-in ASP.NET Core DI container. This has both pros and cons for you as an app developer:
- Pros:

    - *It's simple to add to an existing ASP.NET Core application.* You can easily add Scrutor to an app that's using the built in container. As the same underlying container is used, you can be confident there won't be any unexpected changes in service resolution.
    - *You can use Scrutor with other DI containers*. As Scrutor uses the built-in DI container, and as most third-party DI containers provide adapters for working with ASP.NET Core, you could potentially use both Scrutor and another container in a single application. I can't see that being a common scenario, but it might make migrating an app to use a third-party container easier than moving between two different third-party containers.
    - *It's very likely to remain supported and working, even if the built-in DI container changes*. Hopefully this won't be a concern, but if the ASP.NET Core team make breaking changes to the DI container, Scrutor seems less likely to be affected than third party containers, as it uses the built-in DI container directly, as opposed to providing an alternative container implementation.

- Cons:
  - *Reduced functionality*. As it uses the built-in container, Scrutor will always be limited by the functionality of the built-in container. The built-in container is intentionally kept very simple, and is unlikely to gain significant extra features.

- **Assembly scanning with Scruto:-**

- The Scrutor API consists of two extension methods on IServiceCollection : Scan() and Decorate() . In this post I'm just going to be looking at the Scan method, and some of the options it provides.

- The Scan method takes a single argument: a configuration action in which you define four things:
  - ✓ *A selector* - which implementations (concrete classes) to register
  - ✓ *A registration strategy* - how to handle duplicate services or implementations
  - ✓ *The services* - which services (i.e. interfaces) each implementation should be registered as
  - ✓ *The lifetime* - what lifetime to use for the registrations

```
services.Scan(scan => scan
  .FromCallingAssembly() // 1. Find the concrete classes
    .AddClasses()       //    to register
.UsingRegistrationStrategy(RegistrationStrategy.Skip)
.AsSelf()    // 2. Specify which services they are registered as
.WithTransientLifetime()); // 3. Set the lifetime for the services
```

- So we have something concrete to discuss, lets imagine we have the following services and implementations in an assembly:

```
public interface IService { }
public class Service1 : IService { }
public class Service2 : IService { }
public class Service : IService { }

public interface IFoo {}
```

```
public interface IBar {}
public class Foo: IFoo, IBar {}
```

- The previous Scan() code would register Service1, Service2, Service and Foo as themselves, equivalent to the following statements using the built in container:

```
services.AddTransient<Service1>();
services.AddTransient<Service2>();
services.AddTransient<Service>();
services.AddTransient<Foo>();
```

13.9 Bundling And Minification :-

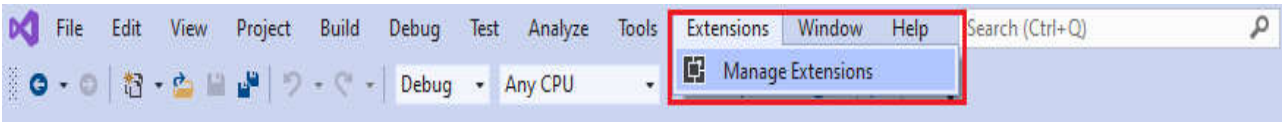
- Bundling is the process of combining multiple files into a single file. We can create CSS, JavaScript and other bundles. Fewer files means fewer HTTP requests and that can improve first page load performance.
- Minification is the process of removing unnecessary data without affecting functionality. It removes comments, extra spaces and converts large variable names to small names.

- Built-in tool **BundlerMinifier :-**

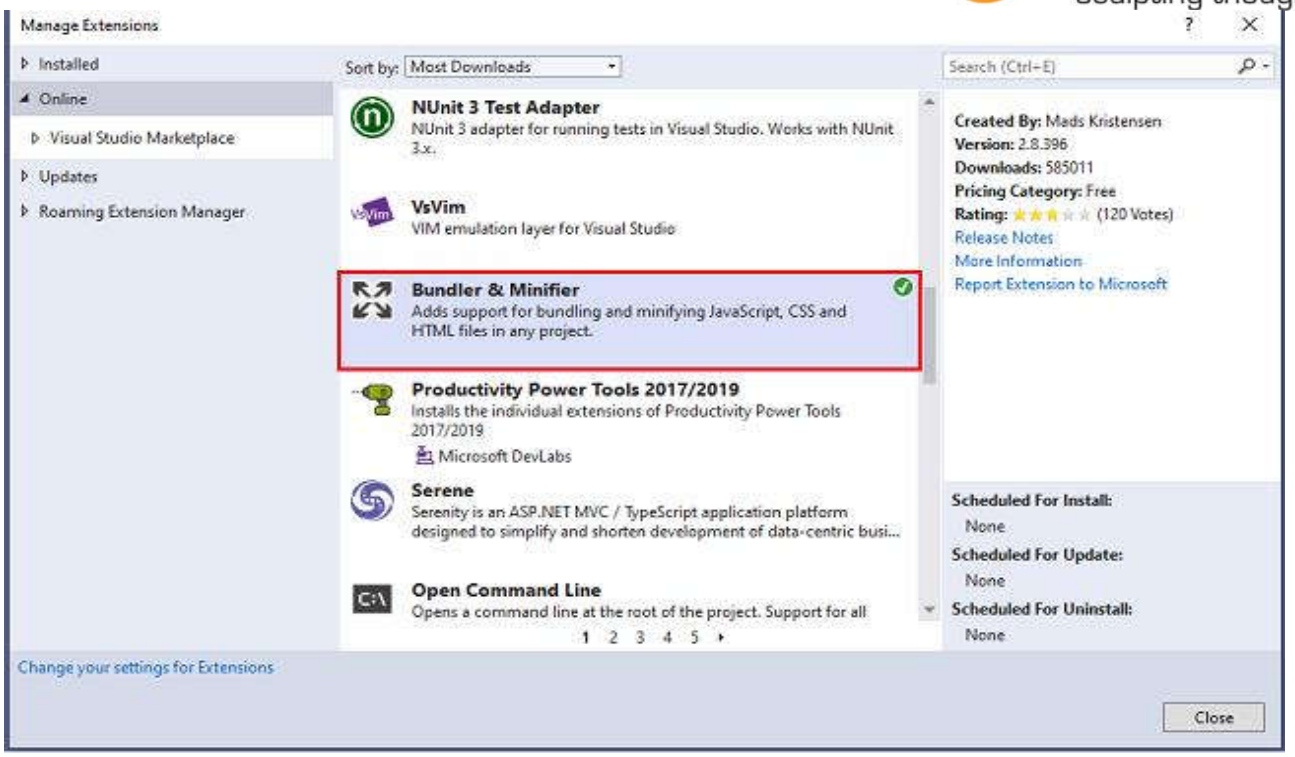
  - BundlerMinifier is the tool built-in to Visual Studio 2017 and 2019 and available as an extension. It's relatively simple, but fully functional, having the capability to integrate into the ASP.NET Core project build process to bundle and minify JavaScript and CSS files.

  - **Steps to add BundlerMinifier :-**

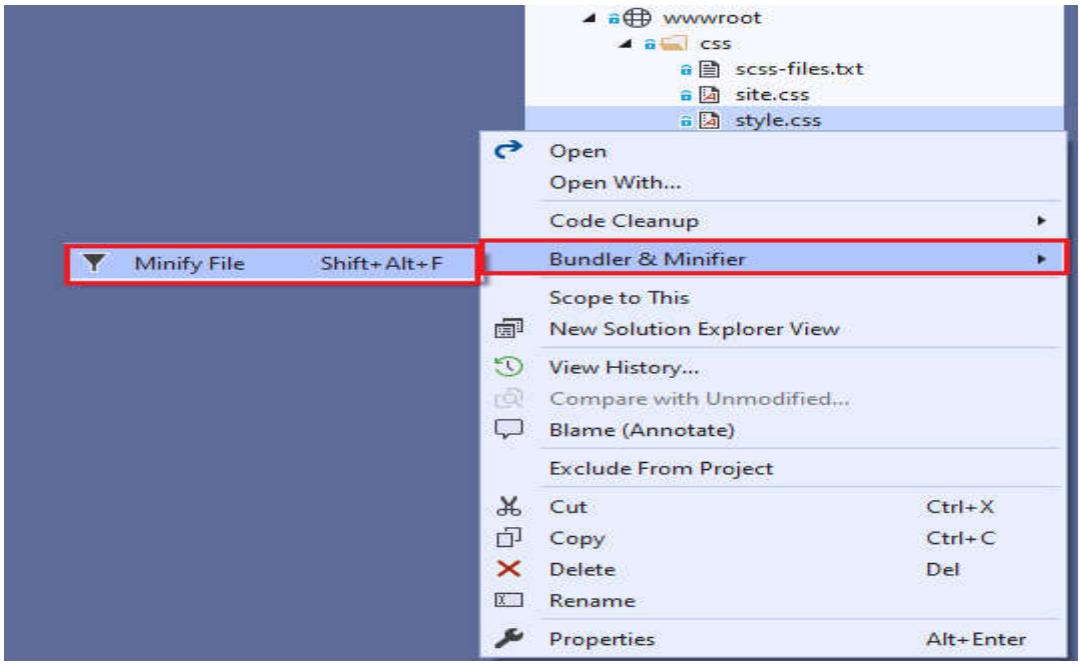
    ✓ In your Visual Studio 2017 or 2019 click on extensions then click on manage extensions. Another window wizard will pop up.

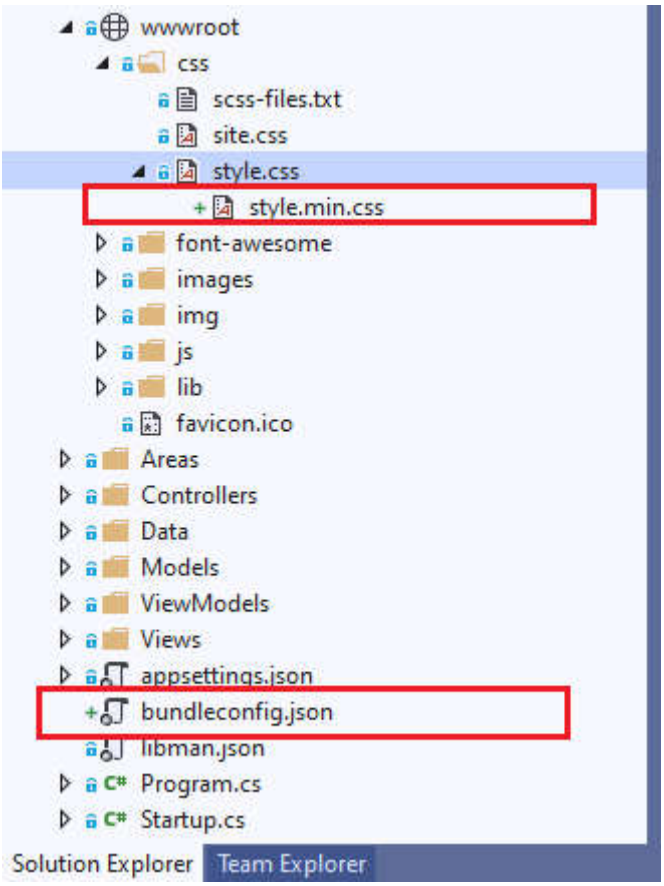


✓ Now click on the right side of the window wizard. In search type bundler & minifier, click on download and install. It will ask you to close Visual Studio then it will start installing. After that it will ask you to modify -- just click on modify and then you are done.



✓ Now start your Visual Studio and open your project. Click on wwwroot folder, select the css file you want minified, then right click and choose bundler & minifier. Then from popup minify file. It will be the same file name with the minified version. Also generate bundleconfig.json file in your project.

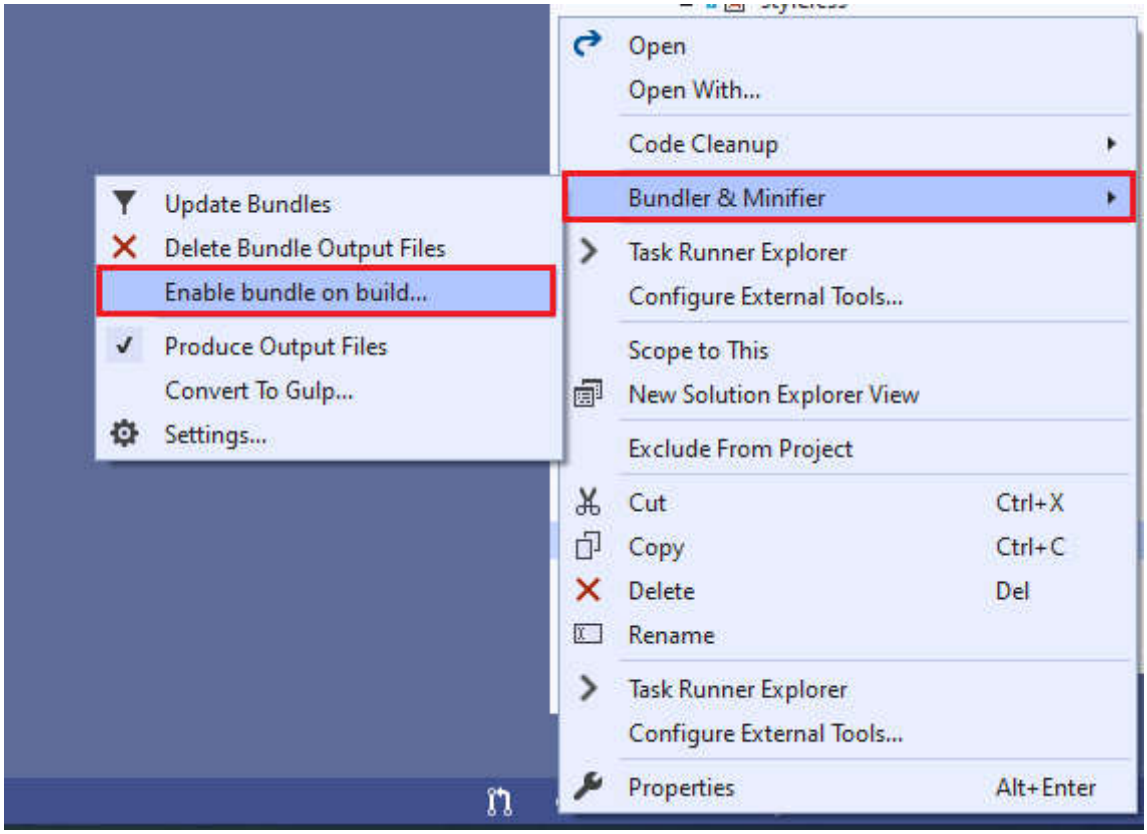




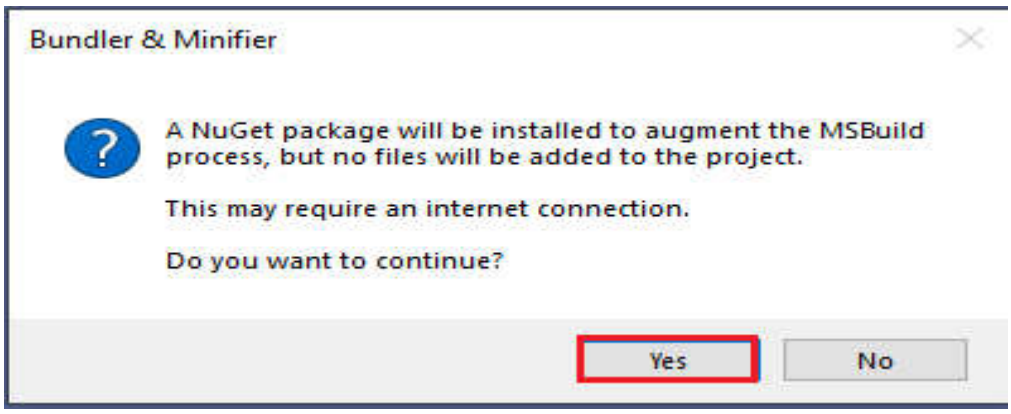
✓ **bundleconfig.json :-**

- The bundleconfig.json file is a standard JSON file and very easy to understand. In this file each bundle is named with the "outputFileName" field and the "inputFiles" to be bundled into that output are simply an array of files. There is only one file in each array here. The bundleconfig.json file also has options to control the minifying process including the option to rename locals for JavaScript files and whether to create a SourceMap file for the JavaScript file.

- If, instead of right-clicking a CSS or JS file, you right click the bundleconfig.json file then you get additional options including easy access to the 'Task Runner Explorer' and an option to 'Enable bundle on build



If you click the 'Enable bundle on build' context menu item then Visual Studio will download an addition NuGet package, if it is not already installed.



13.10 AutoMapper :-

- AutoMapper is used to convert these models to data transfer objects and then the collection of data transfer objects is sent back to the client.

- AutoMapper is a ubiquitous, simple, convention-based object-to-object mapping library compatible with.NET Core. It is adept at converting an input object of one kind into an output object of a different type. You can use it to map objects of incompatible types. You can take advantage of AutoMapper to save the time and effort needed to map the properties of incompatible types in your application manually.

- You can use AutoMapper to map any set of classes, but the properties of those classes have identical names. If the property names don't match, you should manually write the necessary code to let AutoMapper know which properties of the source object it should map to in the destination object.

- To get started you should install AutoMapper, and dotConnect for Oracle package(s) in your project. You can install these packages either from the NuGet Package Manager tool inside Visual Studio or, from the NuGet Package Manager console using the following commands:

```
Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
Install-Package dotConnect.Express.for.Oracle
```

- In Dot Net 5 and latter version:-

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddAutoMapper(typeof(Startup));
   services.AddControllersWithViews();
}
```

- If Dot Net Core 6, Make following changes Program.cs :-

```
builder.Services.AddAutoMapper(typeof(Program));
builder.Services.AddControllersWithViews();
```

- Create Db Class for user :-

```
public class User
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
}
```

- Create ViewModel fro User:-

```
public class UserViewModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```

- **Profiles:-**

  - A good way to organize our mapping configurations is with Profiles. We need to create classes that inherit from Profile class and put the configuration in the constructor.

```
public UserProfile()
{
    CreateMap<User, UserViewModel>();
}
```

  - UserProfile class creates the mapping between our User domain object and UserViewModel. As soon as our application starts and initializes AutoMapper, AutoMapper will scan our application and look for classes that inherit from the Profile class and load their mapping configurations.

- Now, let's define a Controller and use the Auto-Mapping capabilities that we just added:

```
public class UserController : Controller
{
   private readonly IMapper _mapper;

   public UserController(IMapper mapper)
   {
      _mapper = mapper;
   }

   public IActionResult Index()
   {
      // Populate the user details from DB
      var user = GetUserDetails();

      UserViewModel userViewModel =
    _mapper.Map<UserViewModel>(user);

      return View(userViewModel);
   }
 }
```

13.11 Image Viewer Plug-In:-

- **Fancy Box:-**

  - Fancy box saves you time and helps to easily create beautiful, modern overlay windows containing images, iframes, videos or any kind of HTML content.

  - This is the 4th generation of Fancy box and brings lots of fresh features. Both Carousel and Panzoom components are used under the hood and help ensure a best-in-class user experience. In addition, they are easy to integrate with each other.

  - **Features :-**

    - Touch and mobile optimized with swipe, drag and pinch-to-zoom gestures
    - Highly customizable with configuration options, Sass and CSS variables
    - Multiple zoom levels
    - Properly manages focus
    - Closes with Back Button
    - Accessible
    - Smooth, natural animations
    - Multiple instances
    - Widely used, battle-tested
    - Best user and developer experience

- **Declarative:-**
  - Add a data-fancybox attribute to any element to enable Fancybox. Use data-src or href attribute to specify the source of the content.

  - If you wish to customize options, simply overwrite default click
    handler:

```
Fancybox.bind("[data-fancybox]", {
  // Your options go here
});
```

- **Grouping:-**

  - Galleries are created by adding the same attribute data-fancybox value to multiple elements. For example, adding data-fancybox="gallery" attribute to multiple elements, will create a gallery from all these elements.

- **Caption :-**

  - Optionally, use data-caption attribute to show a caption under the content. Additionally, you can use caption option to change its content

```
Fancybox.bind('[data-fancybox="gallery"]', {
  caption: function (fancybox, carousel, slide) {
    return (
      `${slide.index + 1} / ${carousel.slides.length} <br />` +
      slide.caption
    );
  },
});
```

13.12 Tuple Method:-

- Tuple in C# is a **reference-type data structure that allows the storage of items of different data types**. It comes in handy when we need to create an object that can hold items of different data types but we don't want to create a completely new type.

- Tuple initially appeared in .NET framework 4.0 and it can contain up to seven elements plus one optional TRest property as the eighth element – (Tuple<T1,T2,T3,T4,T5,T6,T7,TRest>). The TRest property is mostly for extension purposes and can hold a nested tuple object.

- **How To Use Tuples in C#:-**
    - We can create tuples in two ways.
        - Using the class-based approach
        - using the Tuple Create() method.

1. Using the class-based approach:-

- We can create a Tuple by using the Tuple class constructor and adding the actual elements that we can store in the Tuple within parentheses:

- **Example:-**

```
var tupleWithOneElement = new Tuple<string>("test ");

var tupleWithTwoElements = new Tuple<string, int>("test ", 24);

var tupleWithFiveElements = new Tuple<bool, int, string, decimal,
    string>(true, 3, "CSharp", 23M, "sankuj-rathore");
```

2. **Using the Tuple Create () approach:-**

   - The Tuple class contains a static Create method which returns a tuple object:

   - **Example:-**

```
var tupleCreatedWithExplicitType = Tuple.Create<string>("code-maze");

var tupleCreatedWithNoExplicitType = Tuple.Create("code-maze");

var tupleWithTwoItems = Tuple.Create("code-maze", 24);

var tupleWithFiveItems = Tuple.Create(true, 3, "code-maze", 23M, "blockchain");

var tupleWithEightElements = Tuple.Create(true, "between", "zero", "and", "one" , false, "bezao", 24);
```

13.13 Session Management:-

- HTTP is a stateless protocol. So HTTP requests are independent messages that don't retain user values or app states. We need to take additional steps to manage state between the requests. In this article, we are going to look at various approaches to HTTP state management that we can use in our application.

- **Cookies :-**

  - Cookies store data in the user's browser. Browsers send cookies with every request and hence their size should be kept to a minimum. Ideally, we should only store an identifier in the cookie and we should store the corresponding data using the application. Most browsers restrict cookie size to 4096 bytes and only a limited number of cookies are available for each domain.

- Users can easily tamper or delete a cookie. Cookies can also expire on their own. Hence we should not use them to store sensitive information and their values should not be blindly trusted or used without proper validations.

- We often use cookies to personalize the content for a known user especially when we just identify a user without authentication. We can use the cookie to store some basic information like the user's name. Then we can use the cookie to access the user's personalized settings, such as their preferred color theme.

- **Example:-**

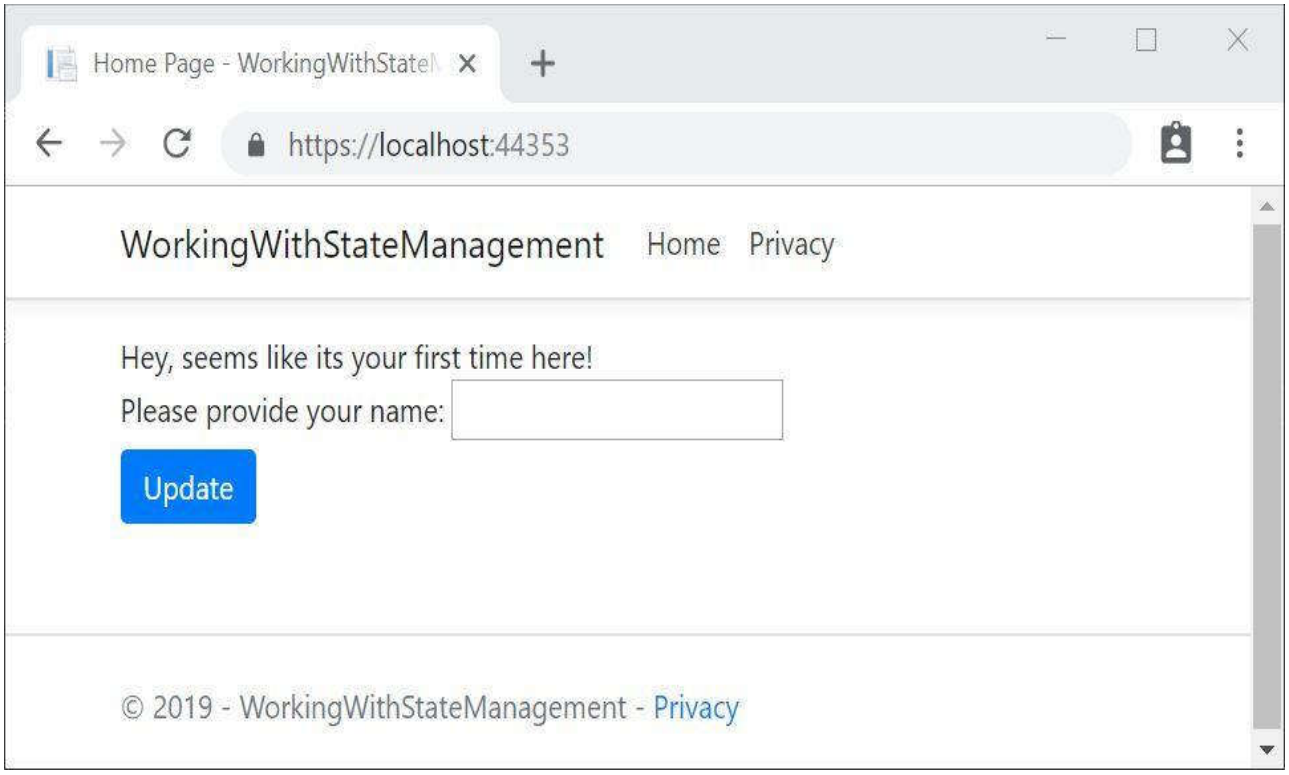
HomeController.cs

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        //read cookie from Request object
        string userName = Request.Cookies["UserName"];
        return View("Index", userName);
    }
    [HttpPost]
    public IActionResult Index(IFormCollection form)
    {
        string userName = form["userName"].ToString();

        //set the key value in Cookie
        CookieOptions option = new CookieOptions();
        option.Expires = DateTime.Now.AddMinutes(10);
        Response.Cookies.Append("UserName", userName, option);
        return RedirectToAction(nameof(Index));
    }
    public IActionResult RemoveCookie()
    {
        //Delete the cookie
        Response.Cookies.Delete("UserName");
        return View("Index");
    }
}
```

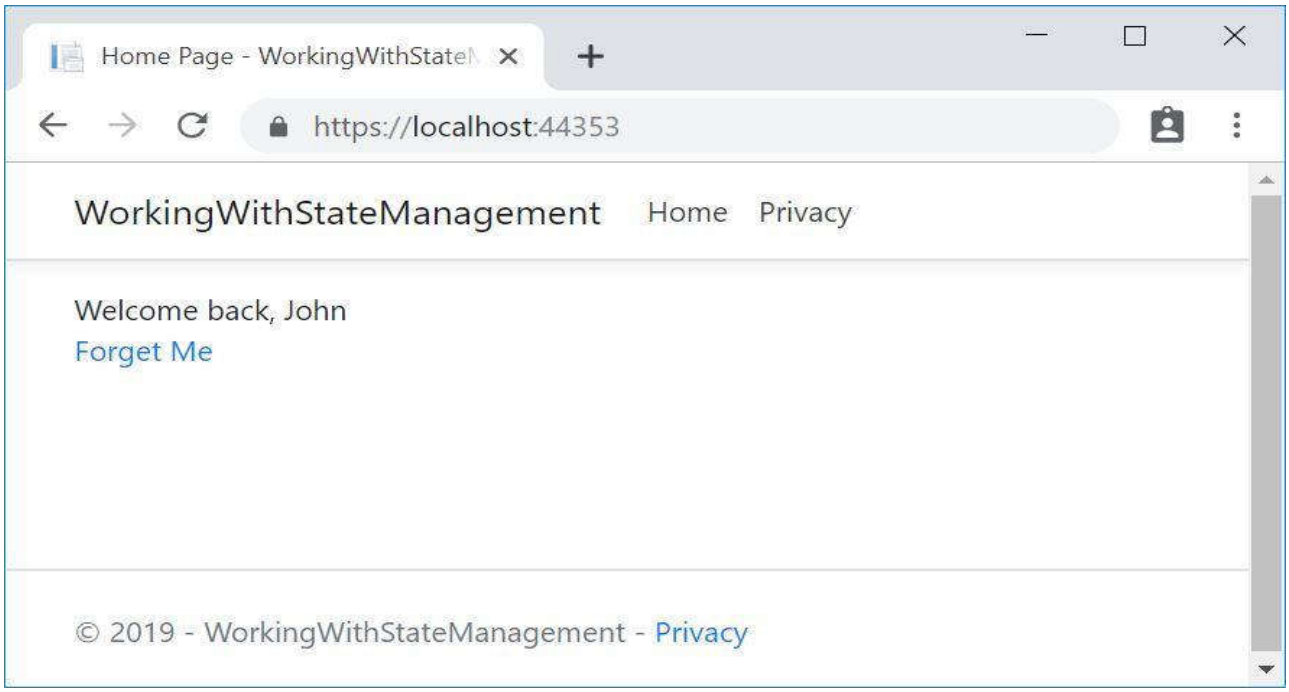
Home.cshtml:-

```
@model string

@{
  ViewData["Title"] = "Home Page";
}

@if (!string.IsNullOrWhiteSpace(Model))
{
  @:<div>Welcome back, @Model</div>
  @Html.ActionLink("Forget Me", "RemoveCookie")
}
else
{
  @:
  <form asp-action="Index">
     <span>Hey, seems like it's your first time here!</span><br />
     <label>Please provide your name:</label>
     @Html.TextBox("userName")
     <div class="form-group">
        <input type="submit" value="Update" class="btn btn-primary" />
     </div>
  </form>
}
```

- Here, we pass the UserName value into the View as the model.
- If the UserName has a value, we greet the user by that name and give the user an option to forget the value by removing it from the cookie.
- In case the UserName is empty, we show an input field for the user to enter his name and a submit button to update this in the cookie.
- Now let's run the application. Initially, the application asks the user to provide a name:

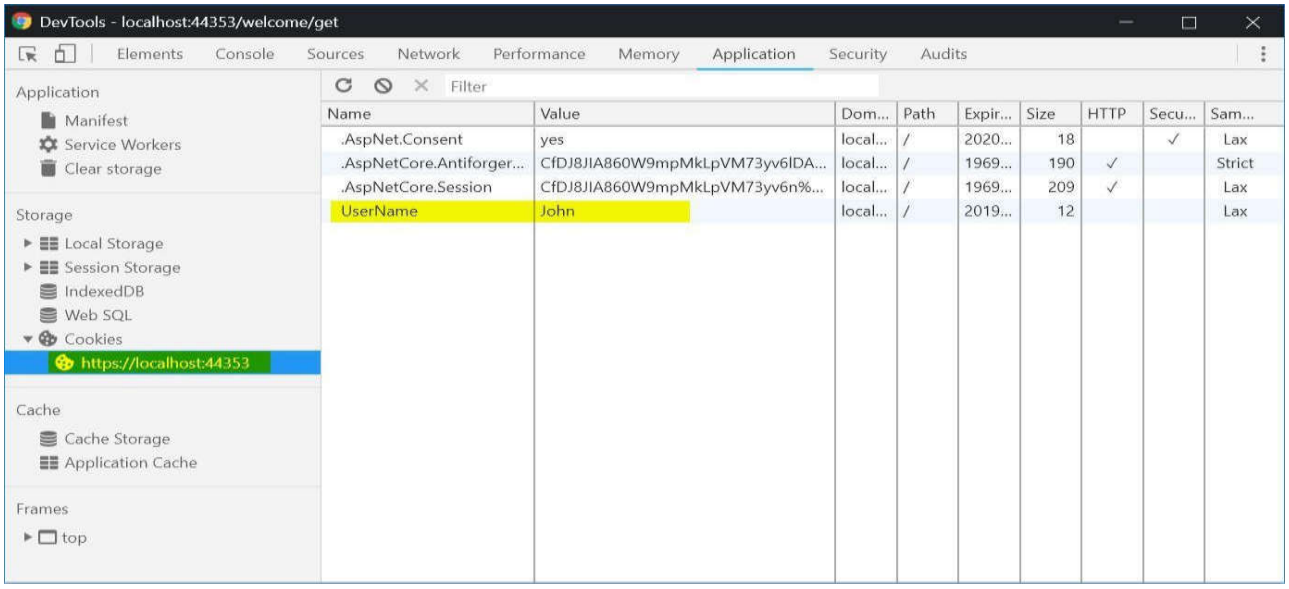


- Once we provide a name and click update, the application greets us:



- Now, where does the application store the cookies? It is stored in the user's browser.

- To inspect a value from a cookie, let's get into the Chrome DevTools window by clicking the F12 key and navigate to the Application tab. We can see that the browser stores cookies for each application:



- **Session State :-**

  - Session state is an ASP.NET Core mechanism to store user data while the user browses the application. It uses a store maintained by the application to persist data across requests from a client. We should store critical application data in the user's database and we should cache it in a session only as a performance optimization if required.

  - ASP.NET Core maintains the session state by providing a cookie to the client that contains a session ID. The browser sends this cookie to the application with each request. The application uses the session ID to fetch the session data.

- While working with the Session state, we should keep the following things in mind:
  - A Session cookie is specific to the browser session
  - When a browser session ends, it deletes the session cookie
  - If the application receives a cookie for an expired session, it creates a new session that uses the same session cookie
  - An Application doesn't retain empty sessions
  - The application retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes
  - Session state is ideal for storing user data that are specific to a particular session but doesn't require permanent storage across sessions
  - An application deletes the data stored in session either when we call the ISession.Clear implementation or when the session expires
  - There's no default mechanism to inform the application that a client has closed the browser or deleted the session cookie or it is expired

- **Example:-**

✓ We need to configure the session state before using it in our application. This can be done in the ConfigureServices() method in the Startup.cs class:

services.AddSession();

✓ Then, we need to enable session state in the Configure() method in the same class:

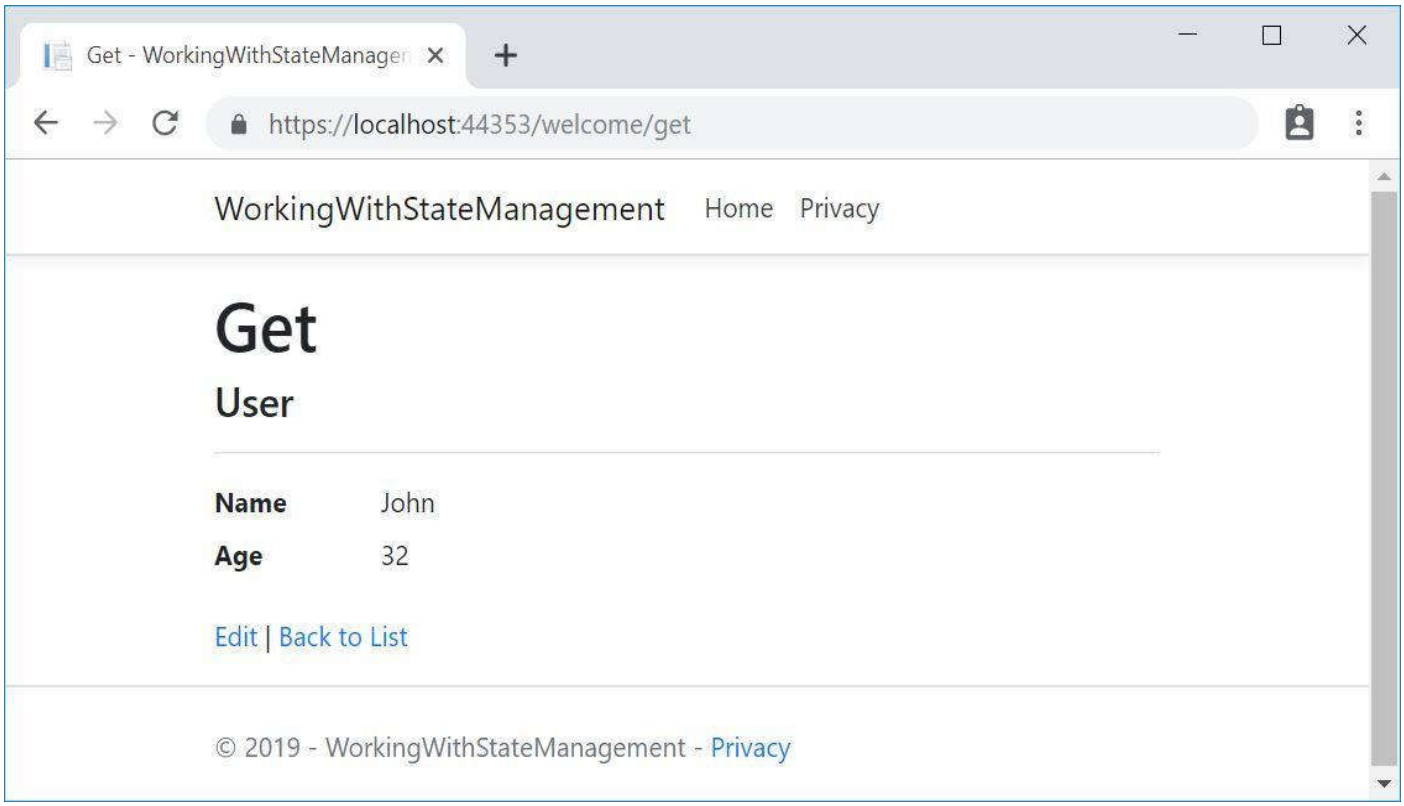
app.UseSession();

- ✓ The order of configuration is important and we should invoke the UseSession() before invoking UseMVC().
- ✓ Let's create a controller with endpoints to set and read a value from the session:

✓ **WelcomeController.cs**

```
public class WelcomeController : Controller
{
    public IActionResult Index()
    {
        HttpContext.Session.SetString("Name", "John");
        HttpContext.Session.SetInt32("Age", 32);

        return View();
    }

    public IActionResult Get()
    {
        User newUser = new User()
        {
            Name = HttpContext.Session.GetString("Name"),
            Age = HttpContext.Session.GetInt32("Age").Value
        };

        return View(newUser);
    }
}
```

- ✓ The Index() method sets the values into session and Get() method reads the values from the session and passes them into the view.
- ✓ Let's auto-generate a view to display the model values by right-clicking on the Get() method and using the "Add View" option.
- ✓ Now let's run the application and navigate to /welcome.
- ✓ This will set the session values.
- ✓ Now let's navigate to /welcome/get:



- **Query String:-**
  - We can pass a limited amount of data from one request to another by adding it to the query string of the new request. This is useful for capturing the state in a persistent manner and allows the sharing of links with the embedded state.
  - Let's add a new method in our Welcome Controller:

```
public IActionResult GetQueryString(string name, int age)
{
    User newUser = new User()
    {
        Name = name,
        Age = age
    };
    return View(newUser);
}
```

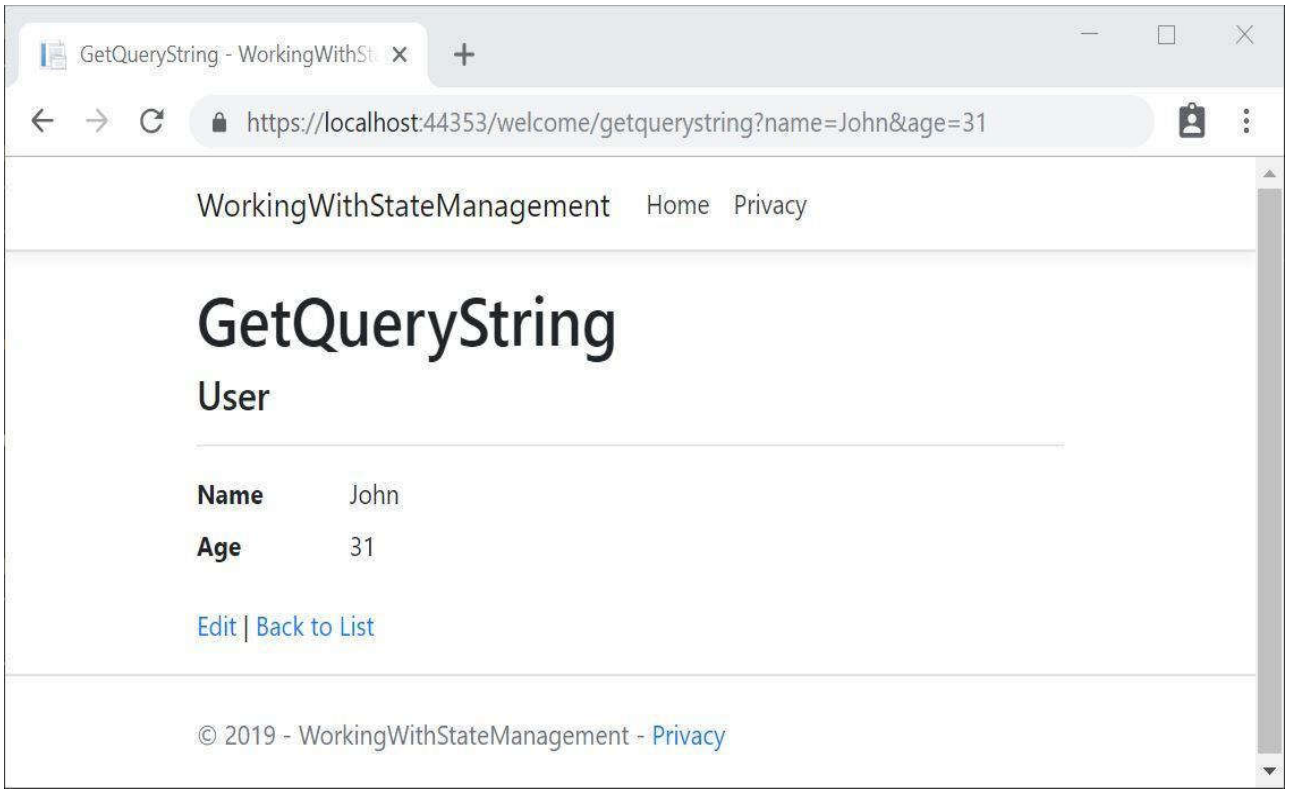
- Model binding maps data from HTTP requests to action method parameters. So if we provide the values for name and age as either form values, route values or query strings, we can bind those to the parameters of our action method.

- For displaying the model values let's auto-generate a view as we did in the previous section.

- We can retrieve both the name and age values from the query string and display it on the page.

- As URL query strings are public, we should never use query strings for sensitive data.

- Now let's invoke this method by passing query string parameters: **/welcome/getquerystring?name=John&age=31**



- **Hidden Fields:-**
  - We can save data in hidden form fields and send back in the next request. Sometimes we require some data to be stored on the client side without displaying it on the page. Later when the user takes some action, we'll need that data to be passed on to the server side. This is a common scenario in many applications and hidden fields provide a good solution for this.
  - Let's add two methods in our WelcomeController:

```
[HttpGet]
public IActionResult SetHiddenFieldValue()
{
   User newUser = new User()
   {
      Id = 101,
      Name = "John",
      Age = 31
   };
   return View(newUser);
}

[HttpPost]
public IActionResult SetHiddenFieldValue(IFormCollection keyValues)
{
   var id = keyValues["Id"];
   return View();
}
```

  - The GET version of theSetHiddenValue() method creates a user object and passes that into the view.
  - We use the POST version of the SetHiddenValue() method to read the value of a hidden field Id from FormCollection.
  - In the View, we can create a hidden field and bind the Id value from Model:

    @Html.HiddenFor(model => model.Id)

- **Temp Data:-**

- ASP.NET Core exposes the TempData property which can be used to store data until it is read. We can use the Keep() and Peek() methods to examine the data without deletion. TempData is particularly useful when we require the data for more than a single request. We can access them from controllers and views.

- TempData is implemented by TempData providers using either cookies or session state.

- **Example :-**

  ✓ Let's create a controller with three endpoints. In the First() method, let's set a value into TempData. Then let's try to read it in Second() and Third() methods:

```
public class TempDataController : Controller
{
    public IActionResult First()
    {
        TempData["UserId"] = 101;
        return View();
    }

    public IActionResult Second()
    {
        var userId = TempData["UserId"] ?? null;
        return View();
    }

    public IActionResult Third()
    {
        var userId = TempData["UserId"] ?? null;
        return View();
    }
}
```

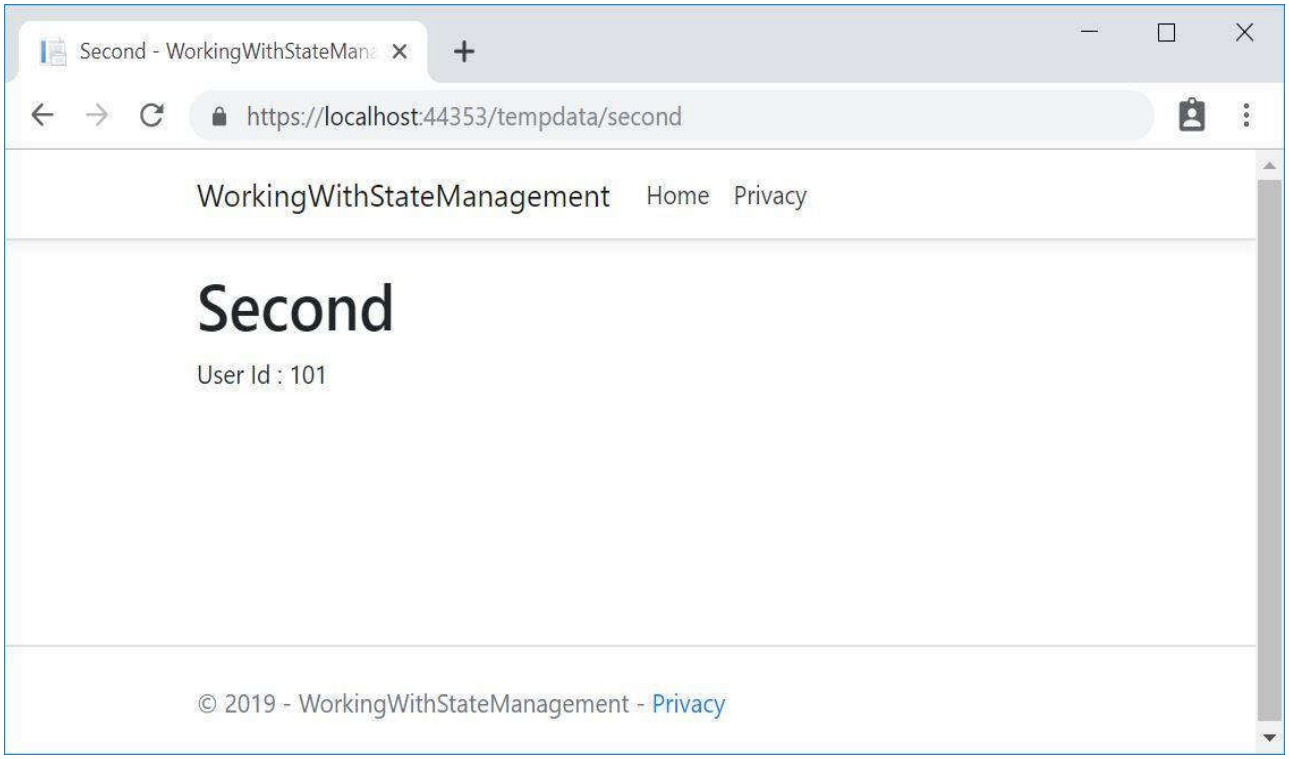
- ✓ Now let's run the application by placing breakpoints in the Second() and Third() methods.
- ✓ We can see that the TempData is set in the First() request and when we try to access it in the Second() method, it is available. But when we try to access it in the Third() method, it is unavailable as is retains its value only till its read.
- ✓ Now let's move the code to access TempData from the controller methods to the views.
- ✓ Let's create a view for the Second() action method:

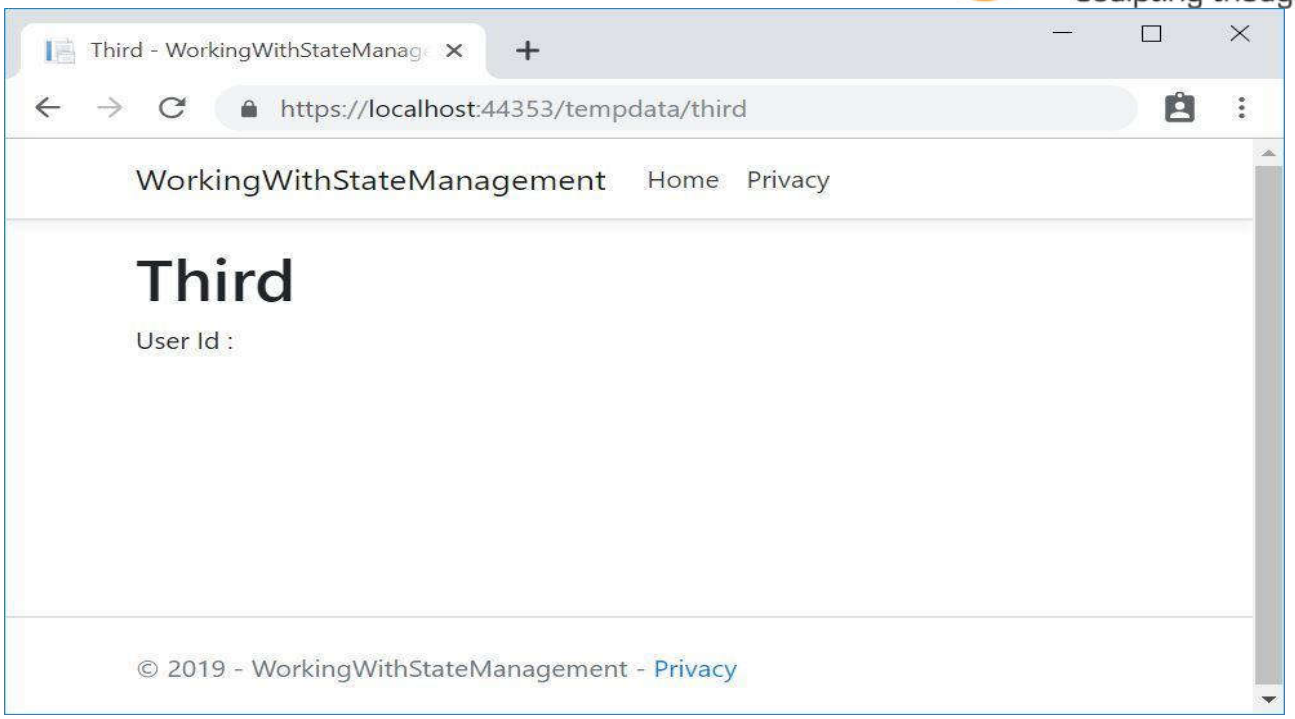
```
@{
    ViewData["Title"] = "Second";
    var userId = TempData["UserId"]?.ToString();
}

<h1>Second</h1>
User Id : @userId
```

- ✓ Similarly, let's create a view for the Third() action method:

```
@{
    ViewData["Title"] = "Third";
    var userId= TempData["UserId"]?.ToString();
}

<h1>Third</h1>
User Id : @userId
```

✓ Let's run the application and navigate to /first, /second and /third





- ✓ We can see that TempData is available when we read it for the first time and then it loses its value. Now, what if we need to persist the value of TempData even after we read it? We have two ways to do that:
  - TempData.Keep()/TempData.Keep(string key) – This method retains the value corresponding to the key passed in TempData. If no key is passed, it retains all values in TempData.
  - TempData.Peek(string key) – This method gets the value of the passed key from TempData and retains it for the next request.
- ✓ Let's slightly modify our second view with one of these methods:

```
var userId = TempData["UserId"]?.ToString();
TempData.Keep();
```

- ✓ Now let's run the application and navigate to /first, /second and /third.
- ✓ We can see that the TempData value persists in the third page even after its read on the second page. Great!

13.14 JWT:-

- Microsoft released .NET 6.0 on November 2021. I have already written couple of articles about JWT authentication on C# Corner. Since .NET 6.0 made some significant changes, I have decided to write one article about JWT authentication using .NET 6.0 version. We will be using Microsoft Identity framework to store user and role information.

- Authentication is the process of validating user credentials and authorization is the process of checking privileges for a user to access specific modules in an application. In this article, we will see how to protect an ASP.NET Core Web API application by implementing JWT authentication. We will also see how to use authorization in ASP.NET Core to supply access to various functionalities of the application. We will store user credentials in an SQL server database, and we will use Entity framework and Identity framework for database operations.

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

    - Header
    - Payload
    - Signature

- Therefore, a JWT typically looks like the following.

    - **xxxx.yyyy.zzzz**

- Implement Jwt Based Authentication in Asp.Net Core review Following Link:-

https://www.c-sharpcorner.com/article/jwt-authentication-and-authorization-in-net-6-0-with-identity-framework

13.15 Broken Image error handler:-

- Regardless of any site you have worked on, there is always a potential problem of a page rendering broken images. This is more likely to happen when images are served from external sources or through accidental deletion within content management platforms.

- The only way I found a way to deal with this issue, is to provide a fallback alternative if the image to be served cannot be found. I've created a FallbackImage() extension method that can be applied to any string variable that contains a path to an image.

- **Example:-**

```
public static class ImageExtensions
{
    /// <summary>
    /// Creates a fallback image if the image requested does not exist.
    /// </summary>
    /// <param name="imageUrl"></param>
    /// <returns></returns>
    public static string FallbackImage(this string imageUrl)
    {
        string cachedImagePath = CacheEngine.Get<string>(imageUrl);

        if (string.IsNullOrEmpty(cachedImagePath))
        {
            string sanitiseImageUrl = string.Empty;

            if (!imageUrl.IsExternalLink())
                sanitiseImageUrl =
$"{HttpContext.Current.GetCurrentDomain()}{imageUrl.Replace("~",
string.Empty)}";
```

```
            // Attempt to request the image.
            WebRequest request = WebRequest.Create(sanitiseImageUrl);

            try
            {
                WebResponse response = request.GetResponse();
                cachedImagePath = imageUrl;
            }
            catch (Exception ex)
            {
                cachedImagePath = "/resources/images/placeholder.jpg";
            }

            // Add image path to cache.
            CacheEngine.Add(cachedImagePath, imageUrl, 5);
        }

        return cachedImagePath;
    }
}
```

- To ensure optimum performance to minimise any unnecessary checks for the same image, the request is stored in cache for 5 minutes.

- The method is using some functionality that I have developed within my own website, which will only work when referenced in your own codebase:

    - **GetCurrentDomain -** get the full URL of the current domain including any protocols and ports.

    - **CacheEngine -** provides a bunch of helper methods to interact with .NET cache provider easily.

13.16 Log4Net:-

- Logging is the heart of an application. It is very important for debugging and troubleshooting, as well as, for smoothness of the application.

- With the help of logging, we can have end-to-end visibility for on-premise systems, to only give a fraction of that visibility for cloud-based systems.

- You can write your logs to a file on a disk or a database, and send an error email.

- **Installation:-**

  ✓ To use Log4Net logging, first you need to add the Log4Net plugin. For adding the plugin, you can do it in two different ways.

    - **Manage NuGet Packages.**
    - **NuGet command.**

  ✓ You can find the required NuGet command for Log4Net below.

  Install-Package log4net -Version 2.0.8

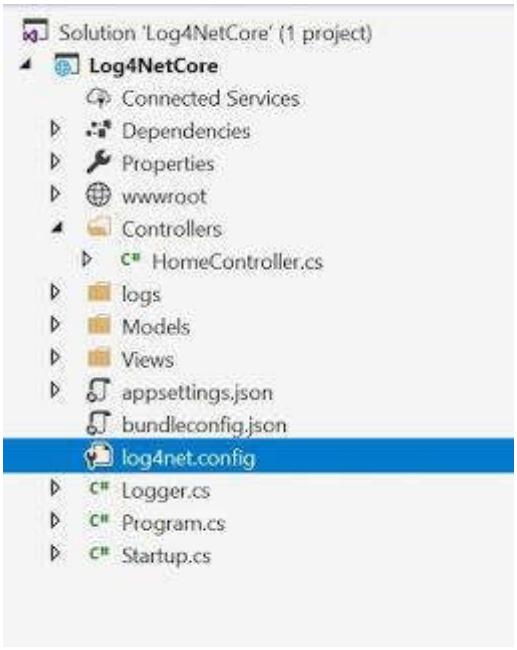
  ✓ **Update Startup file:-**

    - We need to register Log4Net middleware into the startup configure section as below.

```
public void Configure(IApplicationBuilder app,
   IHostingEnvironment env, ILoggerFactory loggerFactory)
{
   loggerFactory.AddLog4Net();
 }
```

✓ **Add log4net.config file:-**

- We need to click on "Add New" to add a file to your project with the name log4net.config.




```
<log4net>
  <appender name="RollingLogFileAppender"
              type="log4net.Appender.RollingFileAppender">
    <lockingmodel type="log4net.Appender.FileAppender+MinimalLock">
    <file value="logs/">
    <datepattern value="yyyy-MM-dd hh.'txt'">
    <staticlogfilename value="false">
    <appendtofile value="true">
    <rollingstyle value="Composite">
    <maxsizerollbackups value="2">
    <maximumfilesize value="15MB">
    <layout type="log4net.Layout.PatternLayout">
      <conversionpattern value="%level  %message  %date">
    </conversionpattern></layout>
```

```
</maximumfilesize></maxsizerollbackups></rollingstyle></appendtofile></staticlogfile
            name></datepattern></file></lockingmodel></appender>
  <root>
    <level value="ALL">
    <appender-ref ref="RollingLogFileAppender">
  </appender-ref></level></root>
</log4net>
```

- Root is neccesary in log4net.config, in which we can define the log level and appender-ref to define appender. For example - FileAppender, ConsoleAppender.

```
<root>
    <level value="ALL">
    <appender-ref ref="RollingLogFileAppender">
     </appender-ref></level>
</root>
```

- **Logging Levels:**

➢ There are seven logging levels.

  - OFF - nothing gets logged (cannot be called)
  - FATAL
  - ERROR
  - WARN
  - INFO
  - DEBUG
  - ALL - everything gets logged (cannot be called)

- **Logging Manager of Log4Net:**

```
        public static class Logger
         {

           private static readonly string LOG_CONFIG_FILE =
@"log4net.config";

           private static readonly log4net.ILog _log =
GetLogger(typeof(Logger));

           public static ILog GetLogger(Type type)
           {
              return LogManager.GetLogger(type);
           }

           public static void Debug(object message)
           {
              SetLog4NetConfiguration();
              _log.Debug(message);
           }

         private static void SetLog4NetConfiguration()
           {
              XmlDocument log4netConfig = new XmlDocument();
              log4netConfig.Load(File.OpenRead(LOG_CONFIG_FILE));

              var repo = LogManager.CreateRepository(
                 Assembly.GetEntryAssembly(),
typeof(log4net.Repository.Hierarchy.Hierarchy));

              log4net.Config.XmlConfigurator.Configure(repo,
log4netConfig["log4net"]);
           }
}
```

- **Load and Read Log4Net Config File:**

```
Private static void SetLog4NetConfiguration()
{
        XmlDocument log4netConfig = new XmlDocument();
        log4netConfig.Load(File.OpenRead(LOG_CONFIG_FILE));

        var repo = LogManager.CreateRepository(
           Assembly.GetEntryAssembly(),
typeof(log4net.Repository.Hierarchy.Hierarchy));

        log4net.Config.XmlConfigurator.Configure(repo,
log4netConfig["log4net"]);
}
```

13.17 AuditLog:-

- An audit trail (also called audit log) is a security-relevant chronological record, set of records, and/or destination and source of records that provide documentary evidence of the sequence of activities that have affected at any time a specific operation, procedure, or event.

- ASP.NET Boilerplate provides the infrastructure to automatically log all interactions within the application. It can record intended method calls with caller info and arguments.

- Related tenant id, caller user id, called service name (the class of the called method), called method name, execution parameters (serialized into JSON), execution time, execution duration (in milliseconds), the client's IP address, the client's computer name and the exception (if the method throws an exception).

- With this information, we not just know who did the operation, but we can also measure the performance of the application and observe the exceptions thrown. Furthermore, you can get statistics about the usage of your application.

- The auditing system uses IAbpSession to get the current UserId and TenantId.

- The Application Service, MVC Controller, Web API and ASP.NET Core methods are automatically audited by default.

- **Configuration:-**

```
public class MyModule : AbpModule
{
public override void PreInitialize()
{
Configuration.Auditing.IsEnabled = false;
}
//...
}
```

- Here are the auditing configuration properties:

1. **IsEnabled**: Used to enable/disable the auditing system completely. Default: true.
2. **IsEnabledForAnonymousUsers:** If this is set to true, audit logs are saved for users that are not logged in to the system. Default: false.
3. **Selectors:** Used to select other classes to save audit logs.
4. **SaveReturnValues:** Used to enable/disable to save return values. Default: false.
5. **IgnoredTypes:** Used to ignore defined types

- Selectors is a list of predicates to select other types of classes that save audit logs. A selector has a unique name and a predicate. The only default selector in this list is used to select application

```
Configuration.Auditing.Selectors.Add(

    new NamedTypeSelector(

        "Abp.ApplicationServices",

        type => typeof (IApplicationService).IsAssignableFrom(type)

    )
```

```
);
```

- You can add your selectors in your module's PreInitialize method. You can also remove the selector above by name if you don't want to save audit logs for application services. This is why it has a unique name (Use simple LINQ to find the selector in Selectors and remove it if you want).

- Note: In addition to the standard audit configuration, MVC and ASP.NET Core modules define configurations to enable/disable audit logging for actions.

- **Enable/Disable by attributes**
  - While you can select auditing classes by configuration, you can use the d and DisableAuditing attributes for a single class or an individual method. Example:Audite

```
[Audited]
public class MyClass
{
    public void MyMethod1(int a)
    {
        //...
    }
    [DisableAuditing]
    public void MyMethod2(string b)
    {
        //...
    }
    public void MyMethod3(int a, int b)
    {
        //...
    }
}
```

14. Use Full Links:-

- MVC pattern
  http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

- Bootstrap
  http://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

- Entity Framework
  https://msdn.microsoft.com/en-us/data/jj591621.aspx