C#.NET

Boxing and UnBoxing in C# -

Boxing -

The process of converting from a value type to a reference type is called boxing. Boxing is implicit conversion.

```
//Boxing
Int anum=123;
Object obj=anum;
Console.WriteLine(anum);
Console.WriteLine(obj);
```

UnBoxing -

The process of converting from reference type to a value type is called unboxing.

```
//Unboxing
Object obj2=123;
Int anum2=(int)obj;
Console.WriteLine(anum2);
Console.WriteLine(obj);
```

Struct	**Class**
The Struct is a value type in C# and it inherits from System.Value Type.	The class is a reference type in C# and it inherits from the System.Object Type.
Struct is usually used for similar amounts of data	Classes are usually used for large amounts of data.
Structs can't be inherited from other types.	Classes can be inherited from other classes.
A structure can't be abstract	Classes can be an abstract type.
Do not have permission to create any default constructor.	We can create default constructor
No need to create an object with a new keyword.	

Difference between Interfaces and Abstract Class in C# -

1. A class can implement any number of interfaces but a subclass can at most use only one abstract class.
2. An abstract class can have non-abstract methods (concrete methods) while in the case of interface, all the methods have to be abstract.
3. An abstract class can declare or use any variables while an interface is not allowed to do so.
4. In an abstract class, all data members or functions are private by default while in an interface all are public, we can't change them manually.

5. In an abstract class, we need to use abstract keywords to declare abstract methods, while in an interface we don't need to use that.
6. An abstract class can't be used for multiple inheritance while the interface can be used for multiple inheritance.
7. An abstract class uses a constructor while in an interface we don't have any type of constructor.

Finalize	**Dispose**
Finalize is used to free unmanaged resources that are not in use, like files, database connections in the application domain and more. These are resources held by an object before that object is destroyed.	Dispose is also used to free unmanaged resources that are not in use like files, database connections in the Application domain at any time.
In the internal process, it is called by Garbage Collector and can't be called manual by user code or any service.	Dispose is exactly called by manual user code.
Finalize belongs to System.Object class	If we need to use the dispose method, we must implement that class via the IDisposable interface.
Implement it when you have unmanaged resources in your code, and make sure that these resources are freed when the Garbage collection happens.	Implement this when you are writing a custom class that will be used by other users.

Sealed Classes in C# -

Sealed classes are used to restrict the inheritance feature of object-oriented programming. Once a class is defined as a sealed class, the class cannot be inherited.

In C#, the sealed modifier is used to define a class as sealed. If a class is derived from a sealed class then the compiler throws an error.

Structs are sealed. You cannot derive a class from a struct.

```
//Sealed Class
Sealed class SealedClass{}
```

Partial Classes in C# -

A partial class is only used to split the definition of a class in two or more classes in the same source code file or more than one source file. You can create a class definition in multiple files, but it will be compiled as one class at run time. Also, when you create an instance of this class, you can access all the methods from all source files with the same object.

Partial classes can be created in the same namespace. It isn't possible to create a partial class in a different namespace. So use the partial keyword with all the class names that you want to bind together with the same name of a class in the same namespace.

```
partial class Class1
 {
     public void Function1()
     {
         Console.WriteLine("Function 1 ");
     }
 }
partial class Class1
{
    public void Function2()
    {
        Console.WriteLine("Function 2 ");
    }
}
 class Program
 {
     static void Main(string[] args)
     {
         Class1 obj = new Class1();
         obj.Function1();
         obj.Function2();
         Console.ReadLine();
     }
 }
```

Nullable reference Types:

Improved static flow analysis that determines if a variable may be *null* before dereferencing it.
Attributes that annotate APIs so that the flow analysis determines *null-state.*
Variable annotations that developers use to explicitly declare the intended *null-state* for a variable.
Exception - System.NullReferenceException - if it dereferenced a variable.
The variable has been assigned to a value that is known to be *not null.*
The variable has been checked against *null* and hasn't been modified since that check.

Encapsulation:

Encapsulation is defined as the process of enclosing one or more items within a physical or logical package. Encapsulation, in object oriented programming methodology, prevents access to implementation details.
Encapsulation is implemented by using access specifiers.
Public- public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class.
Private- private access specifiers allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

Protected- protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance.
Internal- internal access specifier allows a class to expose its member variables and member functions and objects in the current assembly. In other words, any member with an internal access specifier can be accessed from any class or method defined within the application in which the member is defined.
protected internal- The protected internal access specifier allows a class to hide its member variables and member functions, except a child class within the same application. This is also used while implementing inheritance.

Arrays in C# -

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. The position of the last item on an array will total the number of items-1. So if an array has 10 items, the last 10th item is in the 9th position.
In C#, arrays can be declared as fixed-length or dynamic.
A fixed-length array can store a predefined number of items.
A dynamic array does not have a predefined size. The size of the dynamic array increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined.
Let's take a look at simple declaration of arrays in C#. The following code snippet defines the simplest dynamic array of integer types that do not have a fixed size.

```
Int[] intArray;
```

As you can see from the above code snippet, the declaration of an array starts with a type of array followed by a square bracket([]) and the name of the array.

```
Int[] intArray;
intArray=new int[5];
```

Virtual Method in C# -

A virtual method is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overridden in the derived class using the override keyword.
When a method is declared as a virtual method in a base class then that method can be defined in a base class and it is optional for the derived class to override that method. The overriding method also provides more than one form for a method. Hence it is also an example of polymorphism.
When a method is declared as a virtual method in a base class and that method has the same definition in a derived class then there is no need to override it in the derived class. But when a virtual method has a different definition in the base class and the derived class then there is a need to override it in the derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member if no derived class has overridden the member.
Virtual method-

1. By default, methods are non-virtual. We can't override a non-virtual method.
2. We can't use the virtual modifier with static, abstract, private or override modifiers.

Interface:

An interface can be defined using the interface keyword. An interface can contain declarations of methods, properties, indexers and events. However it cannot contain fields, auto-implemented properties.
You can not apply access modifiers to interface members. All the members are public by default. If you use an access modifier in an interface, then the C# compiler will give a compile-time error. "The modifier 'private/public/protected' is not valid for this theme".
An interface can only contain declarations but not implementations.
An interface can be implemented explicitly using <InterfaceName>.<MemberName>.
Explicit implementation is useful when class is implementing multiple interfaces, it is more readable and eliminates the confusion. It is also useful if interfaces have the same method name coincidently.
A class or a struct can implement one or more interfaces using colon(:).
Syntax - <Class or Struct name> : <Interface name>

```
Interface IFile
{
        Void ReadFile();
        Void WriteFile(string text);
}
Class FileInfo: IFile
{
        Public void ReadFile()
        {
                Console.WriteLine("Reading File");
        }
        Public void WriteFile(string text)
        {
                Console.WriteLine("Writing to File");
        }
}
```

1) Interfaces can contain declarations of method, properties, indexers and events.
2) Interface cannot include private, protected or internal members. All the members are public by default.
3) Interfaces can not contain fields, and auto-implemented properties.
4) A class or a struct can implement one or more interfaces implicitly or explicitly. Use public modifier when implementing interface implicitly, whereas don't use it in case of explicit implementation.
5) Implement interface explicitly using interface.MemberName.

6) An interface can inherit one or more interfaces.

Static Class:

1) Static classes cannot be instantiated.
2) All the members of a static class must be static. Otherwise the compiler gives an error.
3) A static class can contain static variables, static methods, static properties, static operators, static events and static constructors.
4) A static class cannot contain instance members and constructors.
5) Indexers and destructors cannot be static.
6) Var cannot be used to define static members. You must specify a type of member explicitly after the static keyword.
7) Static classes are sealed classes and therefore, cannot be inherited.
8) A static class cannot inherit from other classes.
9) Static class members can be accessed using ClassName.MemberName.
10) A Static class remains in memory for the lifetime of the application domain in which your program resides.

Static members in Non-static class:

1) The normal class(non static class) can contain one or more static methods, fields, properties,events and other non-static members.
2) It is more practical to define a non-static class with some static members, than to declare an entire class as static.

Static Fields:

1) Static fields in a non-static class can be defined using the static keyword.
2) Static fields of a non-static class are shared across all the instances. So, changes done by one instance would reflect in others.

Static Methods:

1) Static methods can be defined using the static keyword before a return type and after an access modifier.
2) Static methods can be overloaded but cannot be overridden.
3) Static methods can contain local static variables.
4) Static methods cannot access or call non-static variables unless they are explicitly passed as parameters.

Static Constructors:

1) The static constructor is defined using the static keyword and without using access modifiers public, private or protected.
2) A non-static class contains one parameterless static constructor. Parameterized static constructor are not allowed.
3) Static constructor will be executed only once in the lifetime . So you cannot determine when it will get called in an application if a class is being used at multiple places.
4) A static constructor can only access static members. It cannot contain or access instance members.

Generics:

1) Generic means not specific to a particular data type.

2) C# allows you to define generic classes, interfaces, abstract classes, fields, methods, static methods, properties, events, delegates and operators using the type parameter and without the specific data type. A type parameter is a placeholder for a particular type specified when creating an instance of the generic type.
3) A generic type is declared by specifying a type parameter in an angle brackets after a type name, e.g. TypeName<T> where T is a type parameter.

Generic Class:

Generic classes are defined using a type parameter in an angle brackets after the class name.

```
Class DataStore<T>
{
        Public T Data { get; set; }
}
```

Above, the DataStore is a generic class. T is called type parameter, which can be used as a type of fields properties, method parameters, return types and delegates in the DataStore class. For example, Data is a generic property because we have used a type parameter T as its type instead of specific data type.

It is not required to use T as type parameter. You can give any name to a type parameter. Generally, T is used when there is only one type parameter. It is recommended to use a more readable type parameter name as per requirement like TSession, TKey, TValue etc.

```
Class KeyValuePair<TKey, TValue>
{
        Public TKey key { get; set; }
        Public TValue Value { get; set; }
}
```

Instantiating Generic Class:

You can create an instance of generic classes by specifying an actual type in angle brackets. The following creates an instance of the generic class DataSource.

```
DataStore<string> store = new DataStore<string> ();
```

Example: Generic class

```
DataStore<string> strStore = new DataStore<string>();
strStore.Data = “Hello World!”;
DataStore<int> intStore = new DataStore<int>();
intStore.Data = 100;
KeyValuePair<int, string> kvp1 = new KeyValuePair<int, string>();
kvp1.Key = 100;
kvp1.Value = “Hundred”;
KeyValuePair<string, string> kvp2 = new KeyValuepair<string, string>();
kvp2.Key = “IT”;
kvp2.Value = “Information Technology”;
```

Generic class Characteristics:

1) A generic class increases the reusability. The more type parameters mean more reusable it becomes. However, too much generalization makes code difficult to understand and maintain.
2) A generic class can be a base class to other generic or non-generic classes or abstract classes.

3) A generic class can derived from other generic or non-generic interfaces, classes, or abstract classes.

Generic fields:

A generic class can include generic fields. However, it cannot be initialized.

Example: Generic Field

```
Class DataStore<T>
{
        Public T data;
}
```

Generic Array:

```
Class DataStore<T>
{
        Public T[] data = new T[10];
}
```

Generic Methods:

A method declared with the type parameters for its return type or parameters is called a generic method.

Generic Method:

```
Class DataStore<T>
{
        Private T[] _data = new T[10];
        Public void AddOrUpdate(int index, T item)
        {
                If(index >= 0 && index <10)
                        _data[index] = item;
        }
        Public T GetData(int index)
        {
                if(index >= 0 && index < 10)
                        Return _data[index];
                Else
                        Return default(T);
        }
}
```

Generic Methods:

```
DataStore<string> cities = new DataStore<string>();
cities.AddOrUpdate(0, “Mumbai”);
cities.AddorUpdate(1, “Chicago”);
cities.AddOrUpdate(2, “London”);
DataStore<int> empIds = new DataStore<int>();
empIds.AddOrUpdate(0,50);
empIds.AddOrUpdate(1,100);
empIds.AddOrUpdate(2,200);
```

Generic Methods Overloading:

```
Public void AddOrUpdate(int index, T data){}
Public void AddOrUpdate(T data1, T data2){}
Public void AddOrUpdate<T>(T data1, Udata2){}
```

```
Public void AddOrUpdate(T data){}
```

A non-generic class can include generic methods by specifying a type parameter in angle brackets with the method name.

Generic Method in Non-generic class:

```
Class Printer
{
        Public void Print<T>(T data)
        {
                Console.WriteLine(data);
        }
}
Printer printer = new Printer();
printer.Print<int>(100);
printer.Print(200);
printer.Print<string>("Hello");
printer.Print("World!");
```

C# Generic Collections & Non-Generic Collections:

C# includes specialized classes that store a series of values or objects are called collections.

System.Collections :- namespace contains the non generic collection types.

System.Collections.Generic :- namespace includes generic collection types.

It is recommended to use the generic collections because they perform faster than non-generic collections and also minimize exceptions by compile time errors.

Generic Collections -

1. List<T> :- It grows automatically as you add elements in it.
2. Dictionary<TKey, TValue> :- Contains Key-Value Pair
3. SortedList<TKey, TValue> :- Contains Key-Value pairs. It automatically adds elements in ascending order of keys by default.
4. Queue<T> : values stored in FIFO. Enqueue() method adds values. Dequeue() method to retrieve values from the collection
5. Stack<T> ;- stores the values in LIFO. It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.
6. Hashset<T> :- contains non-duplicate elements. It eliminates duplicate elements.

Non-Generic Collections:

1) Arraylist - stores objects of any type like an array. No need to specify the size of the Arraylist like with an array as it grows automatically.
2) SortedList - Stores key-value pairs. It automatically arranges elements in ascending order of keys by default.
3) Stack - stores values in LIFO. It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.
4) Queue - Stores values in FIFO. It keeps the order in which the values were added. It provides an Enqueue() method to add values and Dequeue() method to retrieve values from the collection.
5) HashTable - Stores Key-value Pairs. It retrieves the values by comparing the hash value of the keys.

6) BitArray - manages the compact array of bit values, which are represented as booleans, where true indicates that the bit is on(1) and false indicates the bit is off(0).

Array and ArrayList:

Array	**ArrayList**
Must include System namespace to use array	Must include System.Collections namespace to use ArrayList.

Array declaration and initialization
Int[] arr = new int[5]
Int[] arr = new int[5]{1,2,3,4,5}
Int[] arr = {1,2,3,4,5}
	ArrayList declaration and initialization
	ArrayList arList=new Arraylist();
	arList.add(1);
	arList.add("Two");
	arlist.add(false);

Array stores a fixed number of elements. The size of an array must be specified at the time of initialization.	Arraylist grows automatically & you don't need to specify the size.
Array is strongly typed. This means that an array can store only specific type of items/elements	ArrayList can store any type of items/elements.
No need to cast elements of an array while retrieving because it is strongly typed and stores a specific type of items only.	The items of arraylist need to be cast to an appropriate data type while retrieving. So boxing and unboxing happens.
Performs faster than arraylist because it is strongly typed.	Performs slow because of boxing and unboxing.
Use a static helper class array to perform different tasks on the array.	Arraylist itself includes various utility methods for various tasks.

Exception Handling in C#:

Exception in the application must be handled to prevent crashing of the program and unexpected result, log exceptions and continue with other functionalities. C# provides built-in support to handle the exception using try, except & finally blocks.

Try Block: Any suspected code that may raise exceptions should be put inside a try{} block. During the execution, if an exception occurs, the flow of the control jumps to the first matching catch block.

Catch block: the catch block is an exception handler block where you can perform some action such as logging and auditing an exception. The catch block takes a parameter of an exception type using which you can get the details of an exception. You can use multiple catch blocks with the different exception type parameters. This is called exception filters. Exception filters are useful when you want to handle different types of exceptions in different ways.

Finally block: the finally block will always be executed whether an exception raised or not. usually, a finally block should be used to release resources, e.g., to close any stream or file objects that were opened in the try block.

```
Try{}
Catch{DivideByZeroException ex}
Catch{FormatException ex}
Finally{}
```

C# Delegates:

The delegate is a reference type data type that defines the method signature. You can define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

Delegate types:

1) Declare a delegate
2) Set a target method
3) Invoke a delegate

A delegate can be declared using the delegate keyword followed by a function signature -

Delegate syntax-

[access specifier] delegate [return type] [delegate name] ([parameters])

Example -

```
public delegate void MyDelegate(String msg);
```

Above, we declared a delegate myDelegate with a void return type and string parameter. A delegate can be declared outside of the class or inside the class. Practically, it should be declared out of the class.

After declaring a delegate, we need to set the target method or lambda function. We can do it by creating an object of the delegate using the new keyword and passing a method whose signature matches the delegate signature.

Example : Set delegate Target -

```
Public delegate void myDelegate(string msg); //declare a delegate
//set target method
MyDelegate del = new MyDelegate(MethodA);
//or
MyDelegate del = MethodA;
//or lambda expression
MyDelegate del= (string msg) => Console.WriteLine(msg);

//target method
Static void MethodA(String message)
{ Console.WriteLine(message);}
```

You can set the target method by assigning a method directly without creating an object or delegate. E.g. MyDelegate del = MethodA.

After setting a target method, a delegate can be invoked using the Invoke() method or using the () operator.Examplee: Invoke delegate

```
del.Invoke("Hello World");
del("Hello World");
```

Example Delegate:

```
Public delegate void MyDelegate(string msg);
Class Program
```

```
{
        Static void Main(string[] args)
        {
                MyDelegate del = ClassA.MethodA;
                InvokeDelegate(del);
                del = ClassB.MethodB;
                InvokeDelegate(del);
                del = (string msg) => Console.WriteLine("Called lambda expression: "+ msg);
                InvokeDelegate(del);
        }
        Static void InvokeDelegate(myDelegate del) //MyDelegate type parameter
        {
                del("Hello World");
        }
}
        class ClassA
        {
                Static void MethodA(string message)
                {
                Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);
                }
        }
        class ClassB
        {
                Static void MethodB(string message)
                {
                        Console.WriteLine("Called ClassB.MethodB() with parameter: "+message);
                }
        }
```

Delegates with Generics

```
        using System;
        public delegate T add<T>(T param1, T param2);
        public class Program
        {
                public static void Main()
                {
                        add<int> sum = Sum;
                        Console.WriteLine(sum(10, 20));
                        add<string> conct = Concat;
                        Console.WriteLine(conct("Hello","World!!"));
                }
                public static int Sum(int val1, int val2)
                {       return val1 + val2;     }
                public static string Concat(string str1, string str2)
                 {
                        return str1 +" "+ str2;
                 }
        }
```

1) Delegate is the reference type data type that defines the signature
2) Delegate type variable can refer to any method with the same signature as the delegate.
3) Syntax: [access modifier] delegate [return type] [delegate name] ([input parameters])
4) A target method's signature must match with the delegate signature.
5) Delegates can be invoked like a normal function or Invoke() method.
6) Multiple methods can be assigned to the delegate using "+" or "+=" operator and removed using "-" or "-=" operator. It is called a multicast delegate.
7) If a multicast delegate returns a value then it returns the value from the last assigned target method.
8) Delegate is used to declare an event and anonymous methods in C#.

Func delegates -

```
Namespace System
{
        Public delegate TResult Func<in T, out TResult>(T args);
}
Class Program
{
        Static int Sum(int x, int y)
        {
                Return x+y;
        }
        Static void Main(string[] args)
        {
                Func<int, int, int> add = Sum;
                Int result = add(10,10);
                Console.WriteLine(result);
        }
}
```

1) Func keyword is a built-in delegate type.
2) Func delegate type must return a value.
3) Func delegate types can have 0 to 16 input parameters.
4) Func delegates do not allow ref and out parameters.
5) Func delegate type can be used with an anonymous method or lambda expression.

Ref Vs Out

Ref	**Out**
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.	A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.

Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter.	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
It is not compulsory to initialize a parameter value before using it in a calling method.	A parameter value must be initialized within the calling method before its use.
When we use REF, data can be passed bi-directionally.	When we use OUT data is passed only in a unidirectional way (from the called method to the caller method).
Both ref and out are treated differently at run time and they are treated the same at compile time.	
Properties are not variables, therefore it cannot be passed as an out or ref parameter.	

Action Delegate -

```
Public delegate void Print(int val);
Static void ConsolePrint(int i)
{
        Console.WriteLine(i);
}
Static void Main(string[] args)
{
        Print prnt = ConsolePrint;
        prnt(10);
}
```

1) Action delegate is same as func delegate except it does not return anything. Return type must be void.
2) Action delegate can have 0 to 16 input parameters.
3) Action delegates can be used with anonymous methods and lambda expressions.

Predicate Delegate-

```
Static bool IsUpperCase(string str)
{
        Return str.Equals(str.ToUpper());
}
Static void Main(string[] args)
{
        Predicate<string> isUpper = IsUpperCase;
        Bool result = isupper("Hello World");
        Console.WriteLine(result);
}
```

1) Predicate delegate takes one input parameter and boolean return type.
2) Anonymous method and Lambda expression can be assigned to the predicate delegate.

Anonymous Method -

An anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type.

```
Example : Anonymous method-
Public delegate void Print(int value);
Static void Main(string[] args)
{
        Int i=10;
        Print prnt = delegate Print(int val)
        {
                val+=i;
                Console.WriteLine("Anonymous method: {0}", val);
        }
        prnt(100);
}
```

1) Anonymous method can be defined using the delegate keyword.
2) Anonymous method must be assigned to a delegate.
3) Anonymous methods can access outer variables or functions.
4) Anonymous method can be passed as a parameter.
5) Anonymous methods can be used as event handlers.

Entity Framework in C# -

Entity framework(hereafter, EF) is the framework ORM(Object-relational-mapping) that Microsoft makes available as part of the .Net development . Its purpose is to abstract the ties to a relational database, in such a way that the developer can relate to the database entity as to a set of objects and then to classes in addition to their properties. In essence, we speak about decoupling between our applications and the logic of data access, which proves to be a major plus. For example: If we need to move-in the context of a single program - to a different manufacturer's database, it would be required to review the way and the instructions with which we interface the data manager on duty.

Entity Framework approaches -

At present, EF mainly allows two types of approaches related to this use. They are Database-First and Code-First. The difference between the two approaches is obvious from their name as with Database-First, we find ourselves in a position, where we have to model a pre-existing database (and therefore, to derive from it our objects), while in the Code-mode First, we will have to prepare by giving them the properties representing the table fields to determine the structure of the database. It is not necessary that Code-First is obliged to work initially in the absence of the database as we can model the classes of an existing database and connect to it to perform the usual operations if I/OR. We can say that the two approaches, beyond some instrumental peculiarity, represent a kind of index of priorities compared to those in power in determining the structure of the data, with which the Application will have to do "before the database" (from which they derived classes) or "before the" code (from which a database model can be textured.)

Entity Framework O/RM considerations-

While EF Core is good at abstracting many programming details, there are some best practices applicable to any O/RM that help to avoid common pitfalls in production apps.

- Intermediate-level knowledge or higher of the underlying database server is essential to architect, debug, profile and migrate data in high performance production apps. For

example, knowledge of primary and foreign keys, constraints, indexes, normalization, DML and DDL statements, data types profiling.

- Functional and integration testing; it's important to replicate the production environment as closely as possible:
    a. Find issues in apps that only show up when using a specific version or edition of the database server.
    b. Catch breaking changes when upgrading EF Core and other dependencies. For example, adding or upgrading frameworks like ASP.NET Core, OData or AutoMapper. These dependencies can affect EF Core in unexpected ways.
- Performance and stress testing with representative loads.The naive usage of some features doesn't scale well. For example, multiple collections includes, heavy use of lazy loading, conditional queries or non-indexed columns, massive updates and inserts with store generated values, lack of concurrency handling, large models, inadequate cache policy.
- Security review -For example, handling of connection strings and other secrets, database permissions for non-deployment operation, input validation for raw SQL, encryption for sensitive data.
- Make sure logging and diagnostics are sufficient and usable. For example, appropriate logging configuration, query tags and application insights.
- Error Recovery- Prepare contingencies for common failure scenarios such as version rollback, fallback servers, scale out and load-balancing, DoS mitigation, and data backups.
- Application deployment and migration. Plan out how migrations are going to be applied during deployment; doing it at application start can suffer from concurrency issues and requires higher permissions than necessary for normal operation. Use staging to facilitate recovery from fatal errors during migration.
- Detailed examination and testing of generated migrations. Migrations should be thoroughly tested before being applied to production data. The shape of the schema and the column types cannot be easily changed once the tables contain production data. For example, on SQL Server nvarchar(max) and decimal(18,2) are rarely the best types for columns mapped to string and decimal properties, but those are the defaults that EF uses because it doesn't have knowledge of your specific scenario.

SQL Server temporal tables-

SQL server temporal tables automatically keep track of all data ever stored in the table, even after that data has been updated. This is achieved by creating a parallel "history table" into which timestamped historical data is stored whenever a change is made to the main table. This allows historical data to be queried, such as for auditing or restored such as for recovery after accidental mutation or deletion.

EF Core now supports-

- The creation of temporal tables using Migrations.
- Transformation of existing tables into temporal tables, again using Migrations.
- Querying historical data.
- Restoring data from some point in the past.

Querying historical data -

EF Core supports queries that include historical data through several new query operators:

- TemporalAsOf- Returns rows that were active (current) at the given UTC time. This is a single row from the current table or history table for a given primary key.
- TemporalAll - returns all rows in the historical data. This is typically many rows from the history table and/or the current table for a given primary key.
- TemporalFromTo - Returns all rows that were active between two given UTC times. This may be many rows from the history table and/or the current table for a given primary key.
- Temporalbetween -The same as TemporalfromTo, except that rows are included that became active on the upper boundary.
- TemporalContainedIn - Returns all rows that started being active and end being active between two given UTC times. This may be many rows from the history table and/or the current table for a given primary key.

.Net Core	**ASP.Net Core**
Open source and cross platform	Open source and cross platform
.Net core is a runtime to execute applications built on it.	ASP.Net core is a web framework to build web apps, IoT apps and mobile backends on the top of .Net Core or .Net Framework.
Install C#.Net core runtime to run applications and install .Net core SDK to build applications.	There is no separate runtime and SDK are available for ASP.Net core. .Net core runtime and SDK includes ASP.Net core libraries.
.Net core 3.1 latest version	ASP.Net Core 3.1 There is no separate versioning for ASP.Net core. It is the same as .Net core versions.

ASP.Net Core - Dependency injection -

Client Class - The client class is a class which depends on the service class
Service Class - The service class is a class that provides service to the client class
Injector class - The injector class injects the service class object into the client class

```
Using System.Linq;
Using System.Text;
Using System.Threading.Tasks;
Namespace DependencyInjection
{
        Class Program
        {
                Static void Main(string[] args)
                {
BusinessLogicService objBusinessLogicService = new BusinessLogicService(new StudentService);
BusinessLogicService objBusinessLogicService = new BusinessLogicService(new TeacherService);
                        Console.ReadLine();
                }
```

```
        }
        //Client Class
        Public class BusinessLogicService
        {
                Private IService iService;
                Public BusinessLogicService(IService _iService)
                {
                        this.iService = _iService;
                        this.iService.GetFirstName();
                        this.iService.GetLastName();
                }
        }
        Public interface IService
        {
                Void GetFirstName();
                Void GetLastName();
        }
        //Service Class
        Public class StudentService: IService
        {
                Public void GetFirstName()
                {
                        Console.WriteLine("Student First Name");
                }
                Public void GetLastName()
                {
                        Console.WriteLine("Student Last Name");
                }
        }
        //Service Class
        Public class TeacherService:IService
        {
                Public void GetFirstName()
                {
                        Console.WriteLine("Teacher First Name");
                }
                Public void GetLastName()
                {
                        Console.WriteLine("Teacher Last Name");
                }
        }
}
```

Solid Principles:

1) Single Responsibility Principle: A class should have one, and only one, reason to change, meaning that a class should have only one job.
2) Open Closed Principle: You should be able to extend a class's behavior, without modifying it.

3) Liskov Substitution Principle: If any module is using a Base class then the reference to that Base class can be replaced with a Derived class without affecting the functionality of the module.
4) Interface Segregation Principle: make fine grained interfaces that are client specific.
5) Dependency Inversion Principle: depend on abstractions not on concrete implementations.

Single Responsibility Principle: One class should be responsible for one task.

```
Class DataAccess
{
        Public static void InsertData()
        {
                Console.Writeline("DB data inserted");
                Console.writeline("Log data inserted");
        }
}
```

So tomorrow if you want to add new logging like event viewer or File I/O then we need to go and change the "DataAccess" class. Which is not right.

```
//Data access class is only responsible for database related operations.
Class DataAccess
{
        Public static void InsertData()
        {
        console.writeline("DB data inserted");
        }
}
//Logger class is only responsible for logging related operations class logger.
Class Logger
{
        Public static void WriteLog()
        {
                console.writeline("Log data inserted");
        }
}
```

Open-Closed Principle: We should strive to write code that doesn't have to be changed every time the requirements change. How we do that can differ a bit depending on the context, such as our programming language.

Create a Base class with Required functionality, and ensure we will not modify that class. (Closed for modification).

Create a derived class by inheriting the Base class for extension.(Open for modification).

```
Public class Rectangle
{
        Public double Width{get; set;}
        Public double Height{get; set;}
}
Public class AreaCalculator
{
        Public double Area(Rectangle[] shapes)
```

```
        {
            Double area=0;
            foreach(var shape in shapes)
            {
                Area += shape.Width*shape.Height;
            }
            Return area;
        }
}
```

If we want to calculate the area of not only rectangles but of circle as well.

```
Public double Area(object[] shapes)
{
        Double area=0;
        foreach(var shape in shapes)
        {
        If (shape is Rectangle)
        {
                Rectangle rectangle=(Rectangle) shape;
                area +=rectangle.Width*rectangle.Height;
        }
        Else
        {
                Circle circle=(Circle) shape;
                area +=circle.radius*circle.radius*math.Pi;
        }
        }
        Return area;
}
```

AreaCalculator isn't closed for modification as we need to change it in order to extend it. Or in other words: it isn't open for extension. In a real world scenario where the code base is ten, a hundred or a thousand times larger and modifying the class means redeploying its assembly/packages.

One way of solving this puzzle would be to create a class for both rectangles and circles as well as any other shapes which defines an abstract method for calculating its area.

```
Public abstract class Shape
{
        Public abstract double Area();
}
Public class Rectangle:Shape
{
        Public double width{get;set;}
        Public double Height{get;set;}
        Public override double Area()
        {
                Return width*height;
        }
}
Public class Circle:Shape
```

```
{
        Public double radius{get;set;}
        Public override double Area()
        {
                Return Radius*radius*Math.Pi;
        }
}
```

As we have moved the responsibility of actually calculating the area away from AreaCalculator's Area method it is now much simpler and robust as it can handle any type of shape that we throw at it.

```
Public double Area(Shape[] shapes)
{
        Double area=0;
        foreach(var shape in shapes)
        {
                area += shape.aera();
        }
        Return area;
}
```

Liskov Substitution Principle: If any module is using a Base class then the reference to that Base class can be replaced with a derived class without affecting the functionality of the module. We must make sure that new derived classes are extending the base classes without changing their behavior. If we are calling a method defined at a base class upon a abstracted class, the function must be implemented properly on the subtype class.

Let's say our system wants to calculate discounts for Enquiries. Now Enquiries are not actual customers they are just leads. Because they are just lends we do not want to save them to database for now.

So we create a new class called as Enquiries which inherits from the "Customer" class. We provide some discounts to the enquiry so that they can be converted to actual customers and we override the "Add" method with an exception so that no one can add an Enquiry to the database.

```
Class Enquiry:Customer
{
        Public override double getDiscount(double TotalSales)
        {
                Return base.getDiscount(TotalSales)-5;
        }
        Public override void Add()
        {
                Throw new Exception("Not allowed");
        }
}
```

So as per polymorphism rule parent "Customer" class object can point to any of it child class objects i.e. "Gold", "Silver" or "Enquiry" during runtime without any issues.

```
List<Customer> Customers=new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new GoldCustomer());
```

```
Customers.Add(new Enquiry());
Foreach(Customer o in Customers)
{
        o.Add(); // thow exception for Enquiry
}
```

As per the inheritance hierarchy the “Customer” object can point to any one of its child objects and we do not expect any unusual behaviour.

But when “Add” method of the “Enquiry” objects is invoked it leads to exception.In other words the “Enquiry” has discount calculation, it looks like a “Customer” but IT IS NOT A CUSTOMER. So the parent cannot replace the child object seamlessly. In other words “Customer” is not the actual parent for the “Enquiry” class. “Enquiry” is a different entity altogether.

```
Interface IDiscount
{
        Double getDiscount(double TotalSales);
}
Interface IDatabase
{
        Void Add();
}
Class Enquiry:IDiscount
{
        Public double getDiscount(double TotalSales);
        {
                Return TotalSales-5;
        }
}
Class Customer:IDiscount, IDatabase
{
        Public virtual void Add()
        {
                //database code goes here
        }
        Public virtual double getDiscount(double TotalSales)
        {
                Return TotalSales;
        }
}
```

In case we make a mistake of adding “Enquiry” class to the list compiler would complain.

```
List<Customer> Customers=new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());  //Error
```

Interface Segregation Principle:

Clients should not be forced to depend upon interfaces that they do not use.

```
Interface IToy
{
        Void setPrice(int price);
        Void setColor(String color);
```

```
        Void move();
        Void fly();
}
Class toyHouse:IToy
{
        Int price;
        String color;
        Public void setPrice(int price)
        {
                this.price=price;
        }
        Public void setColor(String color)
        {
                this.color=color;
        }
        Public void move()
        {
                Throw new exception("move not allowed");
        }
        Public void fly()
        {
                Throw new exception("fly not allowed");
        }
}
```

ToyHouse needs to provide implementation of the move() and fly() methods, even though it does not require them. This is a violation of the Interface Segregation principle. Such violations affect code readability and confuse programmers. Violation of the Interface Segregation Principle also leads to Violation of the complementary Open Closed principle. As an example, consider that the Toy interface is modified to include a walk() method to accommodate toy robots. As a result, you now need to modify all existing toy implementation classes to include a walk() method even if the toys don't walk. In fact the Toy implementation classes will never be closed for modifications, which will lead to a fragile application that is difficult and expensive to maintain.

By following the Interface Segregation Principle, you can address the main problem of the toy building application - The Toy interface forces clients(implementation classes) to depend on methods that they do not use.

The solution is - Segregation the Toy interface into multiple role interfaces each for a specific behavior. Let's segregate the toy interface, so that our application now have three interfaces: toy, Movable, and Flyable.

```
Interface IToy
{
        Void setPrice(int price);
        Void setColor(String color);
}
Interface IMovable
{
        Void move();
```

```
}
Interface IFlyable
{
        Void fly();
}
```

As all toys will have a price and color, all Toy implementation classes can implement this interface. Then, we wrote the movable and flyable interfaces to represent moving and flying behaviors in toys.

```
Class ToyHouse:IToy
{
        Int price;
        String color;
        Public void setPrice(double price)
        {
                this.price=price;
        }
        Public void setColor(String color)
        {
                this.color=color;
        }
}
Class ToyPlane implements IToy, IMovable, IFlyable
{
        Double price;
        String color;
        Public void setPrice(string color)
        {
                this.price=price;
        }
        Public void setColor(String color)
        {
                this.color=color;
        }
        Public void move(){ //code related to moving plane}
        Public void fly(){ // code related to flying plane}
}
```

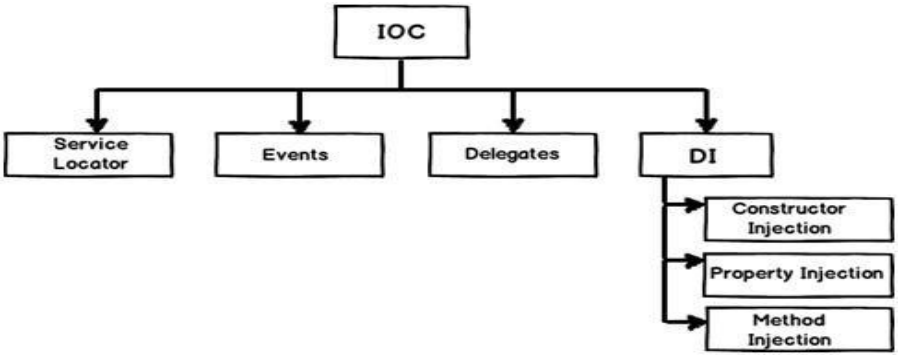
There are times when you might need your interface to have multiple methods, and that's ok. *Include methods which are all very specific to the interface and the client will most likely want to interact with them, therefore packaging them together in the same interface is the right thing to do.*

Dependency inversion Principle:

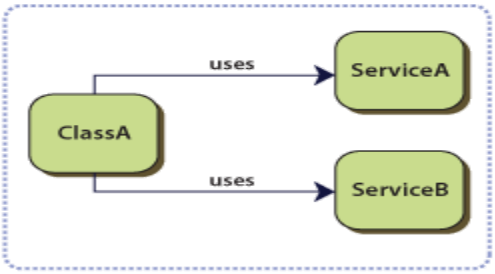
Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module but they should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Inversion of Control: with traditional programming, the main function of an application might make function calls into a menu library to display a list of available commands and query the user to select one. The library thus would return the chosen option as the value of the function

call, and the main function uses this value to execute the associated command. In this interaction, my code is in control: it decides when to ask questions, when to read responses, and when to process those results.
With inversion of control, on the other hand, the program would be written using a software framework that knows common behavioral and graphical elements, such as windowing systems, menus, controlling the mouse, and so on. The custom code “fills in the blanks” for the framework, such as supplying a table of menu items and registering a code subroutine for each item, but it is the framework that monitors the user’s actions and invokes the subroutine when a menu item is selected.
In the command line from I control when these methods are called, but in the window example I don’t. Instead I had control over to the windowing system. It then decides when to call my methods, based on the bindings I made when creating the form. **The control is inverted - it calls me rather me calling the framework. This phenomenon is Inversion of Control. (also known as the hollywood principle -”Don’t call us, we’ll call you”).**
So rather than the internal program controlling the flow, events drive the program flow. Event flow approach is more flexible as there is no direct invocation which leads to more flexibility. You can delegate the control flow by callback delegates, observer pattern, events, DI(Dependency injection) and lot of other ways.



Problem:
You have a classes that have dependencies on services or components whose concrete type is specified at design time. In this example, ClassA has dependencies on ServiceA and ServiceB.



Solution:
1. To replace or update the dependencies, you need to change your classes source code.
2. The concrete implementations of the dependencies have to be available at compile time.
3. Your classes are difficult to test in isolation because they have direct references to dependencies. This means that these dependencies cannot be replaced with stubs or mocks.

4. Your classes contain repetitive code for creating, locating and managing their dependencies.

Any of the following conditions justifies using the solution described in this pattern:

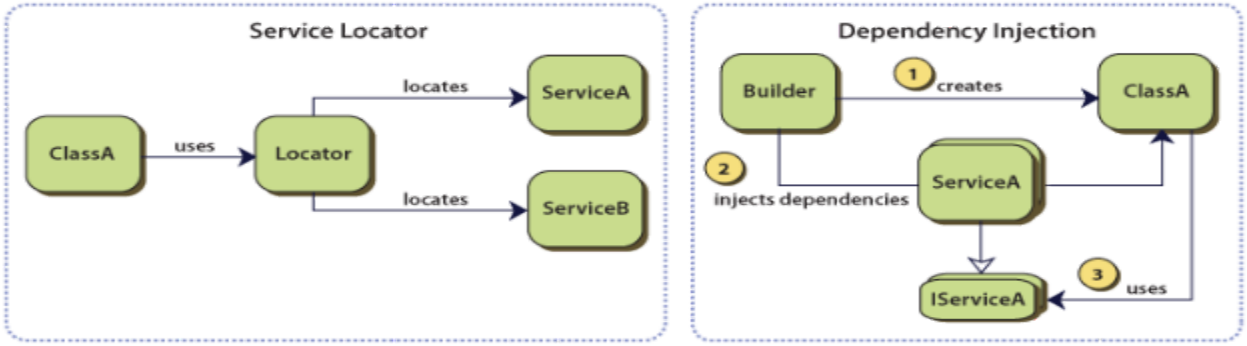
1. You want to decouple your classes from their dependencies so that the dependencies can be replaced or updated with minimal or no changes to your classes source code.
2. You want to write classes that depend on classes whose concrete implementations are not known at compile time.
3. You want to test your classes from being responsible for locating and managing the lifetime of dependencies.

Solution:

Delegate the function of selecting a concrete implementation type for the classes dependencies to an external component or source.

Implementation Details:

The Inversion of control pattern can be implemented in several ways. The Dependency Injection pattern and the Service Locator pattern are specialized versions of this pattern that delineate different implementations. Figure shows the conceptual view of both patterns.



Service Locator:

Create a service locator that contains references to the services and that encapsulates the logic to locate them. In your classes, **use the service locator to obtain service instances.** The service locator does not instantiate the services. It provides a way to **register services and it holds references to the services.** After the service is registered, the service locator can find the service.

```
Public interface IServiceLocator { T GetService<T>(); }
```

Now lets see a very simple implementation of this contract:

```
Class ServiceLocator:IServiceLocator
{
//map that contains pairs of interfaces and references to concrete implementations.
        Private IDictionary<object, object> services;
        Internal ServiceLocator()
        {
                this.services.Add(typeof(IServiceA), new ServiceA());
                this.services.Add(typeof(IServiceB), new ServiceB());
                this.services.Add(typeof(IServiceC), new ServiceC());
        }
        Public T GetService<T>()
```

```
        {
            Try
            {
                Return (T) services[typeof(T)];
            }
            catch(KeyNotFoundException)
            {
                Throw new ApplicationException("The requested service is not registered.");
            }
        }
}
```

The generic GetService() method returns a reference the correct implementation fetching it from the dictionary. This is known as client would invoke the service:

```
IServiceLocator locator = new ServiceLocator();
IServiceA myServiceA = locator.GetService<IServiceA>();
```

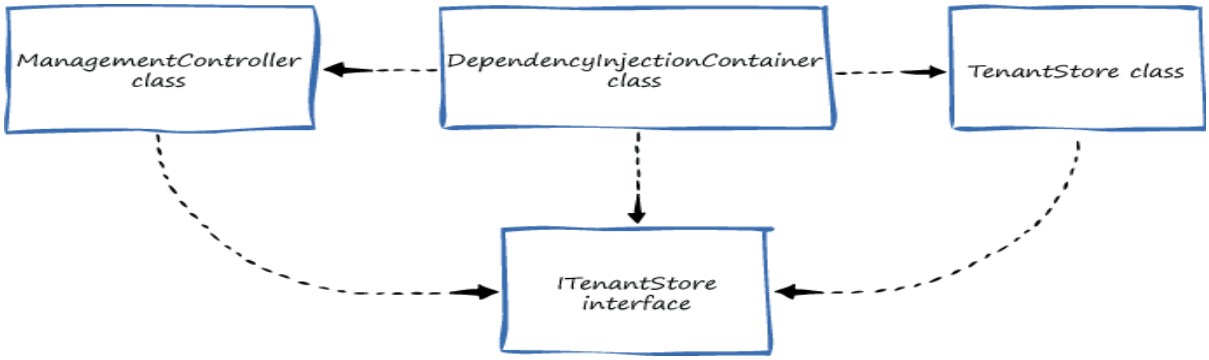
The clients do not know the actual classes implementing the service. They only have to interact with the service locator to get to an implementation.

Dependency Injection:

Declaratively express dependencies in your class definition. Use a **Builder object to obtain valid instances of your object's dependencies and pass them to your object during the object's creation and/or initialization.**

```
Public class ManagementController:Controller
{
    Private readonly ITenantStore tenantStore;
    Public ManagementController(ITenantStore tenantStore)
    {
        this.tenantStore=tenantStore;
    }
    Public ActionResult Index()
    {
        var model = new TenantPageViewData<IEnumerable<string>>
        (this.tenantStore.GetTenantNames())
        { Title="Subscribers" };
        Return this.View(model);
    }
}
```

ManagementController constructor receives an ITenantStore instance as a parameter, injected by some other class. The only dependency in the ManagementController class is on the interface type. This is better because it doesn't have any knowledge of the class or component that is responsible for instantiating the ITenantStore object.



The class that is responsible for instantiating the TenantStore object and inserting it into the ManagementController class is called the DependencyInjection Container class.

Singleton Design Pattern:

1. Ensures a class has only one instance and provides a global point of access to it.
2. A Singleton is a class that only allows a single instance of itself to be created and usually gives simple access to that instance.
3. Most commonly, singleton's doesn't allow any parameters to be specified when creating the instance since the second request of an instance with a different parameter could be problematic (if the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.).
4. There are various ways to implement the Singleton Pattern in C#. The following are the common characteristics of a Singleton Pattern.
    A. A single constructor, that is private and parameterless.
    B. The class is sealed.
    C. A static variable that holds a reference to the single created instance, if any.
    D. A public static means of getting the reference to the single created instance, creating one if necessary.

Example:

```
Namespace Singleton
{
        Class Program
        {
            Static void main(string[] args)
            {
                Console.WriteLine("Singletomn design pattern starts");
                Calculate fromTeachaer = Calculate.GetInstance;
                fromTeachaer.PrintDetails("From Teacher");
                Calculate fromStudent = Calculate.GetInstance;
                fromStudent.PrintDetails("From Student");
                Console.WriteLine("Singletomn design pattern ends");
            }
        }
        public sealed class Calculate
        {
            private static int counter = 0;
```

```
        private static Calculate instance = null;
        public static Calculate GetInstance
        {
            get
            {
                if (instance == null)
                {
                    instance = new Calculate();
                }
                return instance;
            }
        }
        private Calculate()
        {
            Counter++; // second time should not call because it is single instance already used
            Console.WriteLine("Counter Value " + counter.ToString());
        }
        public void PrintDetails(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

==========ASP.NET==========

Validators in ASP.NET:

ASP.Net validation controls define an important role in validating the user input data. Whenever the user gives the input, it must always be validated before sending it across to various layers of an application. If we get the user input with validation, then chances are that we are sending the wrong data. So validation is a good idea to do whenever we are taking input from the user. There are the following types of validation in ASP.Net:

1. Client-Side Validation
2. Server-Side Validation

Client-Side Validation:

When validation is done on the client browser, then it is known as Client-Side Validation. We use JavaScript to do the Client-Side Validation. JavaScript gives us full control over Client-Side Validation. Now Microsoft is also embracing jQuery. Client-Side validation is responsive and quick for end users, but not a secure form of validation. Client-Side Validation is faster, typically looks good , often associated with messages and input.

Server-Side Validation:

When validation occurs on the server, then it is known as Server-Side Validation. Server-Side Validation is a secure form of validation. The main advantage of Server-Side Validation is if the user somehow bypasses the Client-Side Validation, we can still catch the problem on server-side. The following are the Validation Controls in ASP.Net.

- RequestFieldValidator Control

- CompareValidator Control
- RangeValidator Control
- RegularExpressionValidator Control
- CustomFieldValidator Control
- ValidationSummary

PostBack property in ASP.NET -

If we create a web page, which consists of one or more Web Controls that are configured to use AutoPostback (every web control will have their own AutoPostBack property), the ASP.NET adds a special JavaScript function to the rendered HTML page. This function is named _doPostBack(). When called, it triggers a PostBack, sending data back to the Web Server. ASP.NET also adds two additional hidden input fields that are used to pass information back to the server. This information consists of ID of Control that raised the event and any additional information if needed. These fields will empty initially as shown below.

<input type="hidden" name="_EVENTTARGET" id="_EVENTTARGET" value="" />
<input type="hidden" name="_EVENTARGUMENT" id="EVENTARGUMENT" value="" />

The following actions will take place when a user changes a control that has the AutoPostback property set to true.

1. On the client side, JavaScript _doPostBack function is invoked, and the page is resubmitted to the server.
2. ASP.NET re-creates the page object using the .aspx file.
3. ASP.NET retrieves state information from the hidden view state field and updates the controls accordingly.
4. The Page.Load event is fired.
5. The appropriate changes event is fired for the control. (If more than one control has been changed, the order of change event is undetermined.)
6. The Page.PreRender event fires, and the page is rendered(transformed from a set of objects to an HTML page.)
7. Finally the Page.Unload event is fired.
8. The new page is sent to the client.

View State in ASP.Net:

View State is the method to preserve the value of the Page and Controls between round trips. It is a Page-Level State Management technique. View State is turned on by default and normally serializes the data in every control on the page regardless of whether it is actually used during a post-back.

A web application is stateless. That means that a new instance of a page is created every time when we make a request to the server to get the page and after the round trip our page has been lost immediately.

Features of View state:

1. Retains the value of the Control after post-back without using a session.
2. Stores the value of Pages and Control Properties defined in the page.
3. Creates a custom View state provider that lets you store View state information in a SQL Server Database or in another data source.

Advantages of view State:

1. Easy to implement.
2. No server resources are required: The view State is contained in structure within the page load.
3. Enhanced security features: It can be encoded and compressed or unicode implementation.

Session State Management in ASP.Net:

- A new instance of the web page class is created each time the page is posted to the server.
- In traditional Web Programming, all information that is associated with the page, along with the controls on the page, would be lost with each roundtrip.
- The Microsoft ASP.NET framework includes several options to help you preserve data on both a per-page basis and an application-wide basis.These options can be broadly divided into the following two categories.
- Client-Side State Management options
- Server-Side State management options

Client-Side State Management:

- Client-based options involve storing information either in the page or on the client computer.
- Some client-based state management options are:
    1. Hidden fields:
        - A hidden field is a control similar to a TextBox control.
        - A hidden field does not render in a Web browser. A user cannot type anything into it.
        - Hidden fields can be used to store information that needs to be submitted along with the web page, but should not be displayed on the page.
        - Hidden fields can also be used to pass session information to and from forms, transparently.
        - To pass the information from one page to another page, you can invoke the server. Transfer method on the click event of a Button control, as shown in the following example.

```
Protected void Button1.Click(object sender, EventArgs e)
{
        Server.Transfer("Page2.aspx");
}
```

        - The target page can access the passed information by referring to individual controls on the source page, as shown in following example:

```
String name = Request.Form["TextBox1"];
String color = Request.Form["HiddenField1"];
```

    2. View State:
        - Each Web page and controls on the page have a property called ViewState.
        - This property is used to automatically save the values of the Web page and each control on the web page prior to rendering the page.

- The view state is implemented using a hidden form field called _VIEWSTATE.
- This hidden form field is automatically created in every Web page.
- When ASP.NET executes a Web page on the Web server , the values stored in the ViewState property of the page and controls on it are collected and formatted into a single encoded string.
- The encoded string is :
    - Assigned to the Value attribute of the hidden form field, _VIEWSTATE.
    - Sent to the client as part of the Web page.
- During postback of a Web page to itself, one of the tasks performed by ASP.NET is to restore the values in _VIEWSTATE.
- Enabling and disabling view state:
    - By default, the view state is enabled for a Web page and the controls on the Web page.
    - You can enable or disable view state for a page by setting the EnableViewState property of a web page, as shown in the following example:
      <%@ Page language="C#" AutoEventWireup="true" EnableViewState="false" CodeFile="Page1.aspx.cs" inherits="Page1" %>
    - You can enable or disable the view state for a control by setting its EnableViewState property to false.
    - When view state is disabled for a page, the view state for the controls on the page is automatically disabled.

3. Cookies:
    a. Cookies are used to store small pieces of information related to a user's computer such as its IP address, browser type, operating system, and Web pages last visited.
    b. Sent to a client computer along with the page output.

Types of Cookies:

A. Temporary Cookies:
    a. Exists in the memory space of a browser.
    b. Also known as session cookies.
    c. Are useful for storing information required for only a short time.

B. Persistent Cookies:
    a. Are saved as a text file in the file system of the client computer.
    b. Are used when you want to store information for a longer period.
    c. Are useful for storing information required for only a short time.

    Creating Cookies:

    Response.Cookies["userName"].Value="Peter";
    Response.Cookies["userName"].Expires=Datetime.Now.AddDays(2);

    Reading Cookies:

    You can access the value of a cookie using the request built-in object.

    If (Request.Cookies["userName"].Value != null)

```
{
        Label1.Text = request.Cookies["userName"].Value;
}
```

4. Query Strings:
    a. A query string provides a simple way to pass information from one page to another.
    b. Is the part of the URL that appears after the question mark(?) character.
    c. You can pass data from one page to another page in the form of a query string using the Response.Redirect method, as shown in the below example:
    `Response.Redirect("BooksInfo.aspx?Category=fiction&Publisher=Sams");`

Server-Side State Management:

- There are situations where you need to store the state information on the server side.
- Server-Side state management enables you to manage application-related and session-related information on the server.
- ASP.Net provides the following options to manage state at server side:
    1. Application State:
        a. ASP.NET provides application state as a means of storing application-specific information such as objects and variables.
        b. The following describes the information in the application state:
            i. Is stored in a key-value pair.
            ii. Is stored to maintain data consistency between server round trips and among pages.
        c. Application state is created the first time a user accesses any URL resource in an application.
        d. After an application state is created, the application-specific information is stored in it.

    Storing and Reading information in application state:

    You can add application-specific information to an application state by creating variables and objects and adding them to the application state.

    For Example:

    `Application["MyVariable"]="Hello";`

    You can read the value of MyVariable using the following code snippet:

    `stringval=(string) Application["MyVariable"];`

    Removing information from application state:

    You can remove existing object or variable, such as Myvariable from an application state using the following code snippet:

    `Application.Remove(["MyVariable"]);`

    You can also remove all the application state variables and objects by using the following code snippet:

    `Application.RemoveAll();`

    Synchronizing application state:

        a. Multiple pages within an ASP.NET web application can simultaneously access the values stored in an application state, which can result in conflicts and deadlocks.

b. To avoid such situations, the HttpApplicationState class provides two methods, Lock() and Unlock().
c. These methods allow only one thread at a time to access application state variables and objects.

2. Session State:
    a. In ASP.Net, session state is used to store session-specific information for a web application.
    b. The scope of session state is limited to the current browser session.
    c. Session state is structured as a key-value pair for storing session-specific information that needs to be maintained between server round trips and between requests for pages.
    d. Session state is not lost if the user revisits a Web page by using the same browser window.
    e. However, session state can be lost in the following ways:
        i. When the user closes and restarts the browser.
        ii. When the user accesses the same Web page in a different browser window.
        iii. When the session times out because of inactivity.
        iv. When the Session.Abandon() method is called within the Web page code.
    f. Each active ASP.NET session is identified and tracked by a unique 120-bit SessionID string containing ASCII characters.
    g. You can store objects and variables in a session state..
    h. You can add a variable, MyVariable with the value HELLO in the session state using the following code snippet:
    Session[“MyVariable”]=”HELLO”;
    i. You can retrieve the value of the variable, MyVariable, using the following code snippet:
    String val = (string)Session[“MyVariable”];

Catching in ASP.NET:

Currently a majority of websites/portals (web pages) are dynamic. (If I talk about dynamic websites, then it doesn’t mean all the pages of websites are dynamic or will be. The probability of this happening is dependent on the user's perspective and the requirements.).
In very common words I can define dynamic pages as including the following:

- Pages that directly interact with people.
- Communication (On page)
- Any media content
- Any type of graphic interaction.

So, generally these types of pages or webs are called dynamic. Now let's find why we really need caching.

Why Caching:

The process is quite bulky and time-consuming. So to overcome that problem some websites have a creation engine that automatically creates all the pages in one action and directly saves

those pages as a HTML structured page. These HTML pages serve the user depending on their requirements.

Multiple Sorts of Pages:

But, do you still think this will be enough? If your answer is yes, then please think more.

Actually the preceding solution will only work if and only if the requested pages are of the same type. Now think, what will happen if the users request a different sort of page?

In this case your web will be stuck again.

So for dealing with that kind of complex but necessary requirements, ASP.NET provides support for caching. Caching is the hero/heroine in this context that will help us to a great extent.

What a Cache does:

A cache simply stores the output generated by a page in the memory and this saved output (cache) will serve us (users) in the future.

Types of Caching:

1) Page Caching:

   Let's explore the caching of an entire page, first.

   To cache an entire page's output we need to specify a directive at the top of our page, this directive is the @OutputCache Let's figure out a simple demo of it.

   ```
   <%@ OutputCache Duration=5 VaryByParam="ID"  %>
   ```

   Here, in that statement Duration and VarByParam are the two attributes of the OutputCache directive. Now let's explore how they work.

   - Duration Attribute:

     This attribute represents the time in seconds of how long the output cache should be stored in memory. After the defined duration the content stored in memory will be cleared automatically.

   - VarByParam Attribute:

     This is the most important attribute; you can't afford to miss that in the OutputCache directory statement. It generally defines the query string parameters to vary the cache(in memory).

     You can also multiple parameter names too, but for that you need to separate them using semicolon(;).

     You can also specify is as "*". In this case the cached content is varied for all the parameters using the querystring.

     For example:

     ```
     <%@ OutputCache Duration=5 VaryByParam="*" %>
     ```

     In the case of caching pages, some pages can generate different content for different browsers. In that scenario we need to add an additional attribute to our statement for overcoming the preceding problem.

     For example:

     ```
     <%@ OutputCache Duration=5 VaryByParam="ID" VaryByCustom="Browser" %>
     Or
     <%@ OutputCache Duration=5 VaryByParam="*" VaryByCustom="Browser" %>
     ```

2) Fragment Caching:

   In some scenarios we only need to cache only a segment of a page. For example a contact us page in a main page will be the same for all the users and for that there is no need to cache the entire page.

So for that we prefer to use the fragment caching option.

```
<%@ OutputCache Duration=5 varyByParam="None" %>
Or
<%@ OutputCache Duration=5 VaryByParam="None" VaryByCustom="Browser" %>
```

3) Data Caching:

Data caching is slightly different from the 2 other caching types. It's much more interesting to see how data caching actually works.

As we know in C# everything is about classes and objects. So ASP.NET supports data caching by treating them as small sets of objects. We can store objects in memory very easily and use them depending on our functionality and needs, anywhere across the page.

I am inserting string value in the cache as:

```
Cache["Website"] = "CSharpCorner";
```

Now for inserting cache into objects, the insert method of the cache class can be used. This insert method is used as follows:

```
Cache.Insert("Website", strName,
New CacheDependency(Server.MapPath("Website.txt")));
```

What we are missing something:

We missed the Time for the cache (don't forgot to use it), let's provide it.

```
Cache.Insert("Website", strName,
New CacheDependency(Server.MapPath("Website.txt"),
DateTime.Now.Addminutes(5), timespan.zero));
```

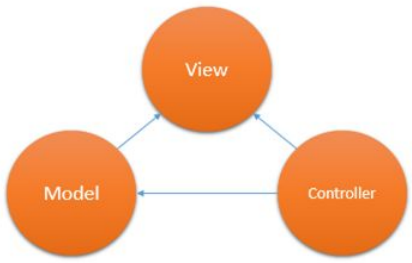
ASP.NET MVC:

model-View-Controller (MVC) is a pattern to separate an application into the following three main concepts

1. Model -
   The Model is the part of the application that handles the logic for the application data. Often model objects retrieve data (and store data) from a database.
2. View -
   The View is the part of the application that handles the display of the data. Most often the views are created from the model data.
3. Controller -
   The Controller is the part of the application that handles user interaction. Typically controllers read data from a view, control user input, and send input data to the model.

The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating web applications. The ASP.NET MVC Framework is a lightweight, highly testable presentation framework that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentications. The MVC framework is defined in the System.Web.Mvc assembly. It provides full control over HTML, JavaScript and CSS. It's the best as well as recommended approach for large-scale applications where various teams are working together.

MVC Design Pattern:



Advantages of an MVC-Based Web Application:

- It makes it very easy to manage complexity by dividing an application into the model, the view and th dee controller.
- It does not use view state or server-based forms.
- Full control over HTML, JavaScript and CSS.
- It provides better support for Test-Driven Development (TDD).
- It works well for Web applications that are supported by large teams of developers and for web designers who need a high degree of control over the application behavior.
- By default support of Facebook and google authentication.
- It is easy to manage a large application by dividing it into multiple areas.
- Multiple View Support - Due to the separation of the model from the view, the user interface can display multiple views of the same data at the same time.
- Change Accomodation - User interfaces tend to change more frequently than business rules(different colors, fonts, screen layouts and levels of support for new devices such as cell phones or PDAs) because the model does not depend on the views, adding new types to the system generally does not affect the model. As a result, the scope of change is confined to the view.
- SoC- Separation of Concerns - Separation of Concerns is one of the core advantages of ASP.NET MVC. The MVC framework provides a clean separation of the UI, Business Logic, Model or Data.
- More Control - The ASP.NET MVC framework provides more control over HTML, JavaScript and CSS than the traditional web forms.
- Testability - ASP.NET MVC framework provides better testability of the web application and good support for test driven development too.
- LightWeight - ASP.NET MVC framework doesn't use View State and thus reduces the bandwidth of the requests to an extent.
- Full features of ASP.NET - One of the key advantages of using ASP.NET MVC is that it is built on top of the ASP.NET framework and hence most of the features of the ASP.NET like membership provides, roles etc can still be used.

ASP.NET MVC Reference namespaces:

1. System.Web.Mvc:-
   Contains classes and interfaces that support the MVC pattern for ASP.NET Web applications. This namespace includes classes that represent controllers, controller factories, action results, views, partial views and model binders.
2. System.Web.Mvc.Ajax:-

Contains classes that support Ajax scripts in an ASP.NET MVC application. The namespace includes support for Ajax scripts and Ajax option settings.

3. System.Web.Mvc.Async:-
   Contains classes and interfaces that support asynchronous actions in an ASP.NET MVC application.
4. System.Web.Mvc.Html:-
   Contains classes that help render HTML controls in an MVC application. The namespace includes classes that support forms, input controls, links, partial views and validation.

Web Forms	**ASP.NET**
Web forms are using Code Behind technique that is divided into two parts .aspx file for View and .aspx.cs/.aspx.vb for Code file.	An ASP.NET MVC web application is a **design pattern** that manages the application into separate 3 folders model, View, Controller.
ASP.NET Web Form has server controls.	ASP.NET MVC has html helpers.
ASP.NET Web Form supports view state for state management at client-side	ASP.NET MVC does not support view state
ASP.NET Web Forms model follows a Page life Cycle.	No Page Life Cycle like web forms. Request cycle is simple in the ASP.NET MVC model.
Provides limited control over HTML, JavaScript and CSS that is necessary in many cases.	Full control over HTML, JavaScript and CSS.
It's good for small scale applications with limited team size.	ASP.NET MVC is a recommended approach for large scale applications where various teams are working together.

MVC Application Life Cycle -

Any web application has two main execution steps, first understanding the request and depending on the type of the request sending out an appropriate response. MVC application life cycle is not different; it has two main phases, first creating the request object and second sending our response to the browser.

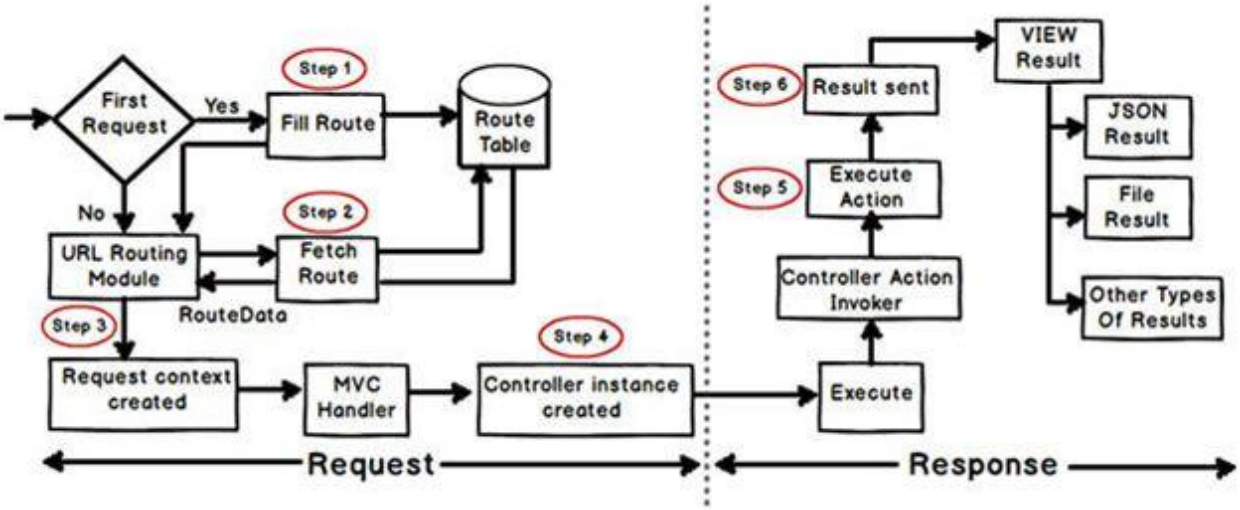
Creating the request object - The request object creation has four major steps. The following is a detailed explanation of the same.

Step 1 - Fill route - MVC requests are mapped to route tables which in turn specify which controller and action to be invoked. So if the request is the first request the first thing is to fill the route table with routes collection. This filling of the route table happens to be the global.asax file.

Step 2- Fetch Route - Depending on the URL sent "UrlRoutingModule" searches the route table to create a "RouteData" object which has the details of which controller and action to invoke.

Step 3- Request context created - The "RouteData" is used to create the "RequestContext" object.

Step 4 - Controller instance created - This request object is sent to “Mvchandler” instance to create the controller class instance. Once the controller class object is created it calls the “Execute” method of the controller class.
Step 5 - Creating a Response object - This phase has two steps executing the action and finally sending the response as a result to the view.



Routing in MVC -

Routing is a mechanism to process the incoming URL that is more descriptive and gives the desired response. In this case, the URL is not mapped to specific files or folders as was the case of earlier days web sites.

There are two types of routing (after the introduction of ASP.NET MVC 5)

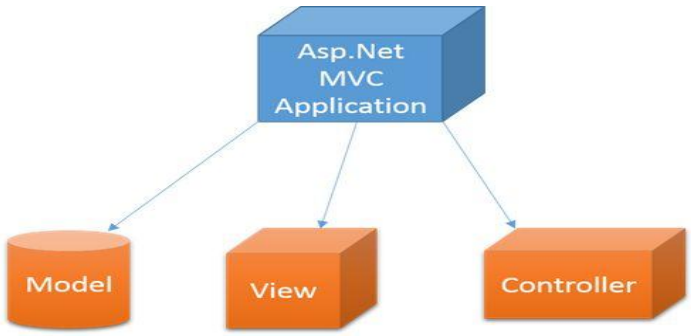
1. Convention-based routing - To define this type of routing, we call the MapRoute method and set its unique name, URL pattern and specify some default values.
2. Attribute-based routing - To define this type of routing, we specify the Route attribute in the action method of the controller.

Routing is the URL pattern that is mapped together to a handler, routing is responsible for incoming browser requests for particular MVC controllers. In other ways let us say routing helps you to define a URL structure and map the URL with a controller. There are three segments for routing that are important.

1. ControllerName
2. ActionMethodName
3. Parameter

ASP.NET MVC Application solution:-

An ASP.NET MVC Application solution is divided into three the minimum folders Model, View and Controller and also we have more folders to place script files and App_start and much more.



1. Model:-
   The model is used to store Data Classes created by LINQ to SQL or Entity framework, or may be the reference of Services from WCF or many more. Finally we just use the model to represent the Data Schema for a View/Partial View.
2. View:-
   The View Folder stores the View pages or Partial View pages for a specific action declared in the Controller Class. Basically the View folder might contain a Shared Folder also in which we can store common pages or user controls that can be used in any controller. Every request for a view or partial view page from an Action method is also checked by the page extension into Shared Folder also.
3. Controller:-
   The Controller is just used to store some Business Logic Class, the controller is a collection of only classes and every class is a child class of the System.Web.Mvc.Controller class. A Controller class only contains some Method known as Action methods that are responsible for returning a View, partial View, Content, JSON Data and more.

Actions:

Actions are only a spatial type of method for writing the code for a specific task and then it is also responsible for returning something to the user and that can be a page/partial page (User Controls). Any action handles the two types of HTTP Request.

1. [HttpGet] -
   [HttpGet] actions to handle requests coming directly from the user and we can also say those requests coming for the first time for an Action method.
2. [HttpPost] -
   [HttpPost] actions are only called when there is a previously view on the client-side and also user submit a HTML Form by a Submit Type button so when that type of action is called and that can also return all the control's values in a **FormCollection** type Object from the HTML Form.
   An Action Method may use many types of return types to return various types of info or values to the user. We have many types to return from an Action method to the user but the maximum time we will use **ActionResult** because its a parent type and any other types can be used in an Action method as a return type.

ActionResult is an abstract class that can have several subtypes.

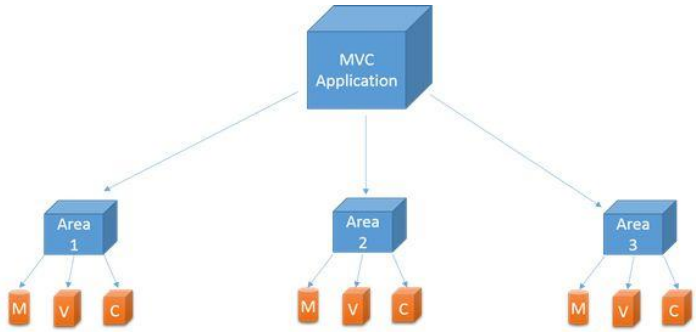
ActionResult Subtypes:-

- ViewResult - Renders a specified view for the user.

- PartialViewResult - Renders a specified partial view to the response stream.
- EmptyResult - An empty response is returned.
- RedirectResult - Performs an HTTP redirection to a specified URL.
- RedirectToRouteResult - Performs an HTTP redirection to a URL that is determined by the routing engine, based on given route data.
- JsonResult - Serializes a given ViewData object to JSON format.
- JavaScriptResult - Returns a piece of JavaScript code that can be executed on the client.
- ContentResult - Writes content to the response stream without requiring a view.
- FileContentResult - Returns a file to the client.
- FileStreamResult - Returns a file to the client, which is provided by a Stream
- FilePatchResult - Returns a file to the client.

Areas in ASP.NET MVC:-

Beginning with ASP.NET MVC 2.0 Microsoft provided a new feature in MVC Applications, Areas. Areas are just used to divide or “isolate” the Modules of a large application in multiple or separated MVC.

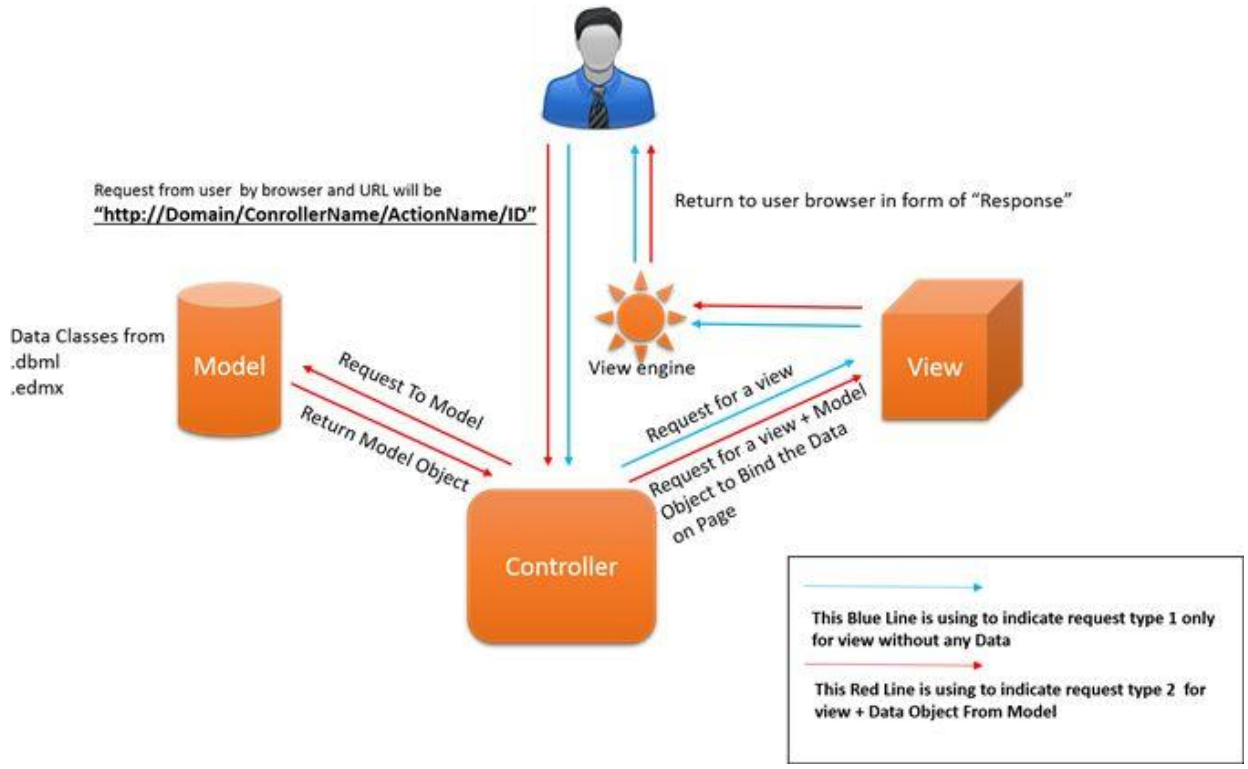


When you add an area to a project, a route for the area is defined in an **AreaRegistration** file. The route sends requests to the area based on the request URL. To register routes, you add code to the **Global.asax** file that can automatically find the area routes in the **AreaRegistration** file.

```
AreaRegistration.RegisterAllAreas();
```

Request Life Cycle for an MVC application:

In an ASP.NET MVC Application there is no page life cycle as in **ASP.NET Web Forms.** Basically a **Request life Cycle** in **MVC Applications.**



In MVC Application when a user makes a request form the browser then the request will be handled by IIS and the request URL will be such as:
Http://SiteName/Controllername/ActionName/ID
In that picture I just described the two types of user requests.

1. Request that returns a view only without data from the Model so when the user requests the request handle by the MVC Handler then it is redirected to the Controller and then it will be directly called an Action Method that is **[HttpGet]**. If that method returns an instance of a **ViewResult** then it will find the suitable view from the View Directory from the web site solution and then the **View Engine (ASPX/RAZOR)** will render the view in HTML for the user's browser.
2. The second Request Type is when a user requests a **View** + data from the **Model** so when the user makes a request that request is handled by the MVC Handler then redirected to the Controller and then it will directly call an Action Method that is **[HttpGet].** If that method returns an instance of **ViewResult** with a model object it will request to the Model class first for an object that we need to use on the view, then we return an instance of **ViewResult "new ViewResult(ModelObject);"** that will be handled by the Model property of the View Class.

There can be two types of relationships between **Views->Model**
Binding

1. Dynamic Binding -
   Dynamic binding is when we pass an object to Viewresult class object to return a view but never define the object type on the view page so we can use that object by reference under the inspection of the DLR but can't use any intelligence on Visual Studio at time of

using this by Model property on the view so we can write a dynamic expression only that will run at run time only.

2. Strongly typed binding -
   Strongly typed binding is when we pass an object to a ViewResult class object to return a view and before that we need to define the type of that object on our view page so we can handle that object in a type safe object and can use that object on our view also the intelligence on Visual Studio.

View Engines -

View Engines are responsible for rendering the HTML from your views to the browser. The view engine template will have different syntax for implementation. Currently there are a few view engines available for MVC and the top view engines are **Razor, ASPX** and ASP.NET also supports some third party View Engines like Spark, NHaml.

- ASPX -
  This **ASPX** is the first view engine for ASP.NET MVC Web Applications. The syntax for writing views with this engine is the same syntax as the ASP.NET Web Forms. We need to use "<%: %>" to write some server side code or if we need to call any object property and all methods view and its parent. We have many view extensions for the pages and all are the same for both server-side languages, either C# or VB like.
  1. .aspx -
     .aspx is an extension of a view page the same as in ASP.NET web sites.
  2. .ascx -
     .ascx is an extension for a partial view in ASP.NET MVC as is a User Control in ASP.NET.
  3. .master -
     .master is an extension for a Master Page the same as in ASP.NET.
- RAZOR -
  The Razor view engine is an advanced view engine from Microsoft for ASP.NET MVC that started with MVC 3 for the first time. Razor uses a "@" character instead of aspx's View Engine "<%: %>" and Razor does not require you to explicitly close the code-block, this view engine is parsed intelligently by the run-time to determine what is a presentation element and what is a code element.
  In my opinion Razor is the best view engine available for MVC since it's much cleaner and easy to write and read. This view engine is compatible with unit testing frameworks.
  This is the default view engine in **MVC 3.0, MVC 4.0 and also in MVC 5.0** for a view page. We have two different page extensions for C#/VB but the same for all types of views whether we are creating a **view, partial view or Layout(Master) pages.**
  - .cshtml -
    When we are using C# as a default language in our ASP.NET MVC Applications for all types of page like View, partial page and layout page.
  - .vbhtml -
    When we are using VB as a default language in our ASP.NET MVC Applications for all types of page like view, partial page and layout page.

Cookies in ASP.NET

Cookies are a State management Technique that can store the values of control after a post-back. Cookies can store user-specific information on the client's machine, such as when the user last visited your site. Cookies are also known by many names, such as HTTP cookies, Browser Cookies, Web Cookies, Session Cookies and so on. Basically cookies are a small text file sent by the web server and saved by the Web Browser on the client's machine.
Cookies are two types-

- Persistent Cookies -
  Persistent Cookies are Permanent Cookies stored as text files in the hard disk of the computer.
- Non-Persistent Cookies -
  Non-Persistent Cookies are temporary. They are also called in-memory cookies and session-based cookies. These cookies are active as long as the browser remains active, in other words if the browser is closed then the cookies automatically expire.

List of properties containing the HttpCookies class.

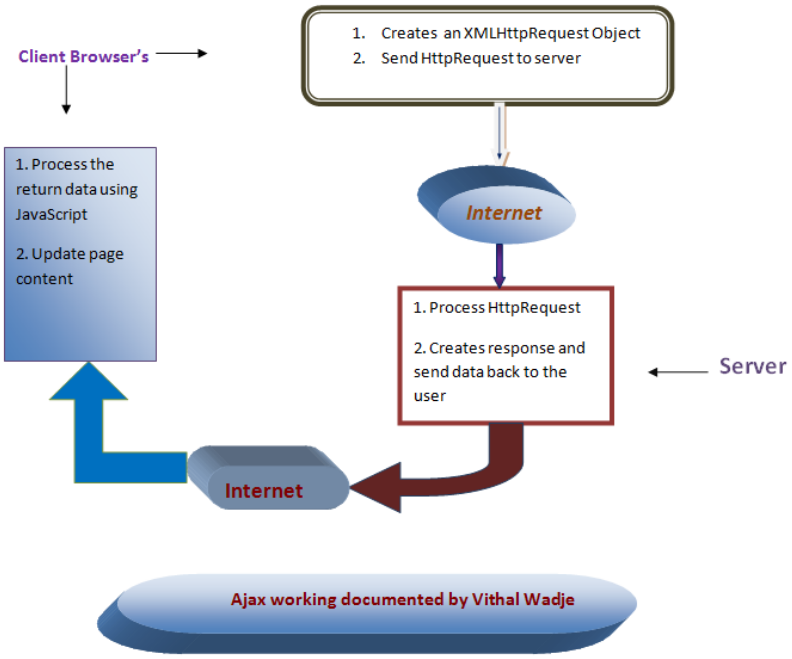
1. Domain - Using these properties we can set the domain of the cookie.
2. Expires - This property sets the Expiration time of the cookies.
3. HasKeys - If the cookies have a subkey then it returns True.
4. Name - Contains the name of the Key.
5. Path - Contains the Virtual Path to be submitted with the Cookies.
6. Secured - If the cookies are to be passed in a secure connection then it only returns True.
7. Value - Contains the value of the Cookies.

Limitation of the Cookies-

1. The size of the cookies is limited to 4096 bytes.
2. A total of 20 cookies can be used in a single website.

Ajax in ASP.NET-

Ajax stands for Asynchronous JavaScript and XML; in other words Ajax is the combination of various technologies such as JavaScripts, CSS, XHTML and DOM etc.
AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page, without reloading the entire page.
We can also define Ajax as a combination of client side technologies that provides asynchronous communication between the user interface and the web server so that partial page rendering occurs instead of complete page post back.
Ajax is platform-independent; in other words AJAX is a cross platform technology that can be used on any Operating System since it is based on XML & JavaScript. It also supports open source implementation of other technologies. It partially renders the page to the server instead of completing the page post back. We use Ajax for developing faster, better and more interactive web applications. Ajax uses a HTTP request between web server and browser.
Using Ajax technologies we can observe in Google Maps, Gmail, Youtube and facebook's tabs.



In the above diagram it's clear how Ajax works, that first depending on the client requests the browser creates the **XMLHttpRequest** and sends it to the server and there after that the server processes the **HttpRequest** given by the browser, processes it and then sends the response back to the user and at the end the browser processes the response given by the server using JavaScript and update the requested content of the particular page instead of the entire page contents.

Ajax is based on Internet Standards -

Ajax is based on Internet standards, the uses a combination of a XMLHttpRequest object to exchange data asynchronously with a server, JavaScript and DOM to interact with the information and CSS is used to style the data and XML is often used as the format for transferring data.

Let us know about the basic information each technology uses in Ajax

1. XHTML and CSS -
   XHTML or HTML is used for providing the markup tags, as used in any typical web site. In addition we utilize CSS for extra styling functionality in relation to presentation and layout. XHTML is a stricter and more standardized form of HTML, which follows the rules of XML such as the requirement for being well-formatted and valid against a schema or DTD.
2. Document Object Model (DOM) -
   The Document Object module is a platform and language independent standard object model for representing HTML or XML.
3. XML -
   XML is typically used as the format for transferring data between the server and the client. Using XML we can represent any applicable data object structure we might wish to represent.
4. XMLHttpRequest (XHR) and JavaScript -

The XMLHttpRequest is the core of the ajax model; without it the model would not exist. The XMLHttpRequest JavaScript object is the enabling technology which is used to exchange data asynchronously with the web server . In short, XMLHttpRequest lets us use JavaScript to make a request to the server and process the response without blocking the user. Naturally, as we are using this JavaScript object, the providing technology is javaScript and hence some knowledge of JavaScript is required to get Ajax applications to function.

- In Ajax **Client** and **Server** communication is done with the help of **HttpRequest**.
- Ajax applications are browser and platform independent.

Ajax and ASP.NET Framework -

ASP.NET Ajax integrates client script libraries with the ASP.NET 2.0 development framework. This new web development technology extends ASP.NET, offering the interactive user interface benefits of Ajax with a programming model that is more familiar to ASP.NET developers, making it very easy to add Ajax to your applications quickly and with minimal effort.

Power of Ajax -

- With Ajax, when a user clicks a button, you can use JavaScript and DHTML to immediately update the UI, and spawn an asynchronous request to the server to fetch results.
- When the response is generated, you can then use JavaScript and CSS to update your UI accordingly without refreshing the entire page. While this is happening, the form on the user's screen doesn't flash, blink, disappear or stall.
- The power of Ajax lies in its ability to communicate with servers asynchronously, using a XMLHttpRequest object without requiring a browser refresh.
- Ajax essentially puts JavaScript technology and the XMLHttpRequest object between your web form and the server.

Everything happens behind the scenes with a minimum request and response cycle without the knowledge of the user.

Advantages of Ajax based application-

- Improved application performance by reducing the amount of data downloaded from the server.
- Rich, Responsive and Slick UI with no page flickers.
- Eliminates frequent page refresh which usually happens in a typical request/response model (Everything is updated on fly).
- Easy to implement as there are a variety of Ajax implementations available around.
- Ajax mechanism works behind the scene, nothing much required from user perspective.
- Works with all browsers.
- Avoids the round trips to the server.
- Rendering of web pages faster.
- Decreases the consumption of server resources.
- Response time of the application is very fast.
- Rendering of data is dynamic.

Using Ajax Extension-

The following are the most commonly used Ajax controls in an ASP.NET application which comes with the ASP.NET framework and available under the Ajax Extension tab of ASP.NET Toolbox present at the left hand side of Microsoft Visual Studio Framework.
These Controls are -

1. ScriptManager -
   When we use any Ajax control then there is a requirement to use the ScriptManager to handle the Scripting on the client side; without the ScriptManger Ajax controls are not run. So it's a requirement to use the ScriptManager.
2. UpDatePanel -
   Update panel is one of the most commonly used Ajax controls which is responsible for updating the particular requested content of the page instead of the entire page which is not requested. The ASP.NET controls which are kept under the update panel will be updated when the user clicks on a particular ASP.NET Control which is used in an application. You can use multiple update panels on a single web page.
3. Timer -
   The timer is also one of the important controls; by using it we can update the particular content of the page automatically without clicking the browser refresh button. The timer control button is used along with the Update Panel so the Contents put under the update panel are automatically updated according to the timing set under the timer_click event.
4. Updateprogress -
   This control is used to notify the user to wait until the requests are processed on the server. Update progress control is used when the user clicks on any tab or control of an application. At that time the progress bar is shown which is the interval between the times taken by the server to process the client request.
5. ScriptManagerProxy -
   When you need to reference a service from your content page and yet the ScriptManager resides on the Master Page use a ScriptMangerProxy works by detecting the main ScriptManager on your page at runtime and hooking itself to that ScriptManger, making sure that any references given to it are also given to the real ScriptManager.
6. Pointer -
   This Ajax Control used to specify the style of the mouse pointer such as arrow, thumb, and progress bar and much more. The above is the basic introduction about the Ajax Extension controls which are available by default in a Microsoft Visual Studio framework.

Web Services in ASP.NET -

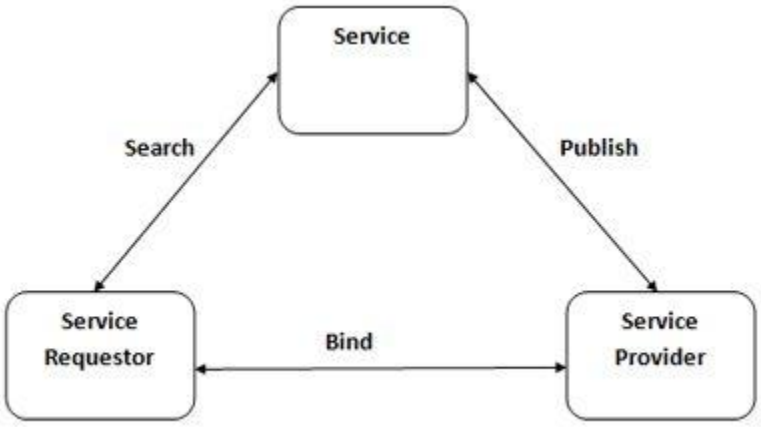
A Web Service is a software program that uses XML to exchange information with other software via common internet protocols. In a simple sense, Web Services are a way of interacting with objects over the Internet, A Web Service is -

- Language Independent
- Protocol Independent
- Platform Independent
- It assumes a stateless service architecture
- Scalable (e.g. multiplying two numbers together to an entire customer-relationship management system).

- Programmable (encapsulates a task)
- Based on XML (open, text-based standard)
  Self-describing (metadata for access and use)
- Discoverable(Search and locate in registries) - ability of applications and developers to search for and locate desired Web services through registries. This is based on UDDI.

Key Web Service technologies -

- XML - Describes only data. So, any application that understands XML-regardless of the application's programming language or platform-has the ability to format XML in a variety of ways (well-formed or valid)
- SOAP - Provides a communication mechanism between services and applications.
- WSDL - Offers a uniform method of describing web services to other programs.
- UDDI - Enables the creation of searchable Web services registries.



Advantages of ASP.NET -

ASP.NET provides service to allow the creation, deployment and execution of Web Applications and Web Services like ASP. ASP.NET is a server-side technology. Web Applications are built using Web Forms. ASP.NET comes with built-in Web Form controls, which are responsible for generating the user interface. They mirror typical HTML widgets such as text boxes or buttons. If these controls do not fit your needs, you are free to create your own user controls.

Advantages of ASP.NET -

- Separation of Code from HTML.
- Support from compiled languages
- Use services provided by the .NET framework.
- Graphical Development Environment
- Update files while the server is running
- XML-Based Configuration Files.

Concepts of Globalization and localization in .NET -

Localization means "the process of translating resources for a specific culture", and Globalization means "the process of designing applications that can adapt to different cultures."

- Proper Globalization Your Application should be able to Accept, Verify and Display all kinds of global data. It should also be able to operate over this data, accordingly.

- Localizability and localization Localizability stands for clearly separating the components of culture based operations regarding the user interface, and other operations from the executable code.

.NET framework has greatly simplified the task of creating the applications targeting the clients of multiple cultures. The namespaces involved in creation of globalize, localizing applications are,

- System.Globalization
- System.Resources
- System.Text

Web.config file in ASP -

Configuration file is used to manage various settings that define a website. The settings are stored in XML files that are separate from your application code. In this way you can configure settings independently from your code. Generally a website contains a single Web.config file stored inside the application root directory. However there can be many configuration files that manage settings at various levels within an application. Usage of configuration file ASP.NET configuration system is used to describe the properties and behaviors of various aspects of ASP.NET applications. Configuration files help you to manage the settings related to your website. Each file is an XML file (with the extension .config) that contains a set of configuration elements. Configuration information is stored in XML-based text files.

Benefits of XML-based configuration files-

- ASP.NET Configuration system is extensible and application specific information can be stored and retrieved easily. It is human-readable.
- You need not restart the web server when the settings are changed in configuration files. ASP.NET automatically detects the changes and applies them to the running ASP.NET application.
- You can use any standard text editor or XML parser to create and edit ASP.NET configuration files.

AppDomain concept in ASP.NET -

ASP.NET introduces the concept of an Application Domain which is shortly known as AppDomain. It can be considered as a Lightweight process which is both a container and boundary. The .NET runtime uses an AppDomain as a container for code and data, just like the operating system uses a process as a container for code and data. As the operating system uses a process to isolate misbehaving code, the .NET runtime uses an AppDomain to isolate code inside of a secure boundary.

The CLR can allow the multiple .Net applications to be run in a single AppDomain.

The CLR isolates each application domain from all other application domains and prevents the configuration, security, or stability of a running .Net application from affecting other applications. An AppDomain can be destroyed without affecting the other AppDomains in the process.

Multiple AppDomains can exist in the Win32 process, the main aim of the AppDomain is to isolate applications from each other and the process is the same as the working of the operating system process. This isolation is achieved by making sure that any given unique virtual address space runs exactly one application and scopes the resources for the process or application domain using that address space.

Win32 processes provide isolation by having distinct memory addresses. The .Net runtime enforces AppDomain isolation by keeping control over the use of memory. All memory in the AppDomain is managed by the run time so the runtime can ensure that AppDoamins Do not access each other's memory.

How to create AppDomain?

AppDomains are generally created by hosts for example Internet Explorer and ASP.NET. The following is an example to create an instance of an object inside it and then execute one of the object's methods. This is the explicit way of creating AppDomain by .NET Applications.

AppDomains are created using the CreateDomain method. AppDomain instances are used to load and execute assemblies(Assembly). When an AppDomain is no longer in use, it can be unloaded.

```
Public class MyAppDomain: MarshalByRefobject
{
        Public string GetInfo()
        {
            Return AppDomain.CurrentDomain.FriendlyName;
        }
}
Public class MyApp
{
        Public static void Main()
        {
            AppDomain adp = AppDomain.CreateDomain(“Rajendra Doman”);
            MyAppDomain apdinfo = (MyAppDomain)apd.CreateInstanceAndUnwrap
            (Assembly.getCallingAssembly() .getName().Name, “MyAppDomain”);
        }
}
```

The AppDomain class implements a set of events that enable applications to respond when an assembly is loaded, when an application domain will be unloaded, or when an unhandled exception is thrown.

Advantages -

A single CLR operating system process can contain multiple application domains. There are advantages to having application domains within a single process.

1. Lower system cost - many application domains can be contained within a single system process.
2. Each application domain can have different security access levels assigned to them, all within a single process.
3. Code in one AppDomain cannot directly access code in another AppDomain
4. The application in an AppDomain can be stopped without affecting the state of another AppDomain running in the same process.
5. An exception on AppDomain will not affect other AppDomains or crash the entire process that hosts the AppDomains.

Query String in ASP.NET -

A QueryString is a collection of characters input to a computer or web browser. A Query String is helpful when we want to transfer a value from one page to another. When we need to pass content between HTML pages or aspx Web Forms in the context of ASP.NET, A Query String is

Easy to use and the Query appears after this separating character, usually a Question Mark (?). It is basically used for identifying data appearing after this separating symbol. A Query String Collection is used to retrieve the variable values in the HTTP query string. If we want to transfer a large amount of data then we can't use the Request.Querystring. Query Strings are also generated by form submission or can be used by a user typing a query into the address bar of the browsers. Syntax of the Query String Request.QueryString(variable)[(index).count]

Advantages-

- Simple to implement
- Easy to get information from Query String
- Used to send or read cross domain(from different domain)

Disadvantages-

- Human Readable
- Client browser limit on URL length
- Cross paging functionality makes it redundant
- Easily modified by the end user.

Master page in ASP.NET -

The extension of MasterPage is '.master'. MasterPage cannot be directly accessed from the client because it just acts as a template for the other Content Pages. In a MasterPage we can have content either inside ContentPlaceHolder or outside it. Only content inside the ContentPlaceHolder can be customized in the Content Page. We can have multiple masters in one web application. A MasterPage can have another MasterPage as Master to it. The MasterPageFile property of a webform can be set dynamically and it should be done either in or before the Page_PreInit event of the WebForm.Page.MasterPageFile = "MasterPage.master".

The dynamically set Masterpage must have the ContentPlaceHolder whose content has been customized in the WebForm. A master page is defined using the following code, <%@ master language="C#" %> Adding a MasterPage to the project.

1. Add a new MasterPage file (MainMaster.master) to the web application.
2. Change the Id of ContentPlaceHolder in <Head> to "cphHead" and the Id "ContentPlaceHolder1" to "cphFirst".
3. Add one more ContentplaceHolder (cphSecond) to MasterPage.
4. To add the master page add some header, footer some default content for both the content placeholders.

```
<form id="form1" runat="server"> Header…
        <br />
        <asp:ContentPlaceHolder id="cphFirst" runat="server"> This is the First Content Place holder (Default) </asp: ContentPlaceHolder>
        <br />
        <asp:ContentPlaceHolder ID="cphSecond" runat="server">This is the second Content Place Holder (Default)</asp:ContetPlaceHolder>
        <br /> Footer…
</Form>-
```

Some Points about Master Pages -

1. The extension of Master Page is '.master'.
2. MasterPage cannot be directly accessed from the client because it just acts as a template for the other Content pages.

3. In a Masterpage we can have content either inside ContentPlaceHolder or outside it. Only Content inside the ContentPlaceHolder can be customized in the Content Page.
4. We can have multiple masters in one web application.
5. A Masterpage can have another MasterPage as Master to it.
6. The content page content can be placed only inside the content tag.
7. Controls of MasterPage can be programmed in the MasterPage and content page but a content page control will never be programmed in MasterPage.
8. A MasterPage of one web application cannot be used in another web application.
9. The MasterPageFile property of a webform can be set dynamically and it should be done either in or before the Page_PreInit event of the WebForm. Page.MasterPageFile=”MasterPage.master”. The dynamically set master page must have the ContentPlaceHolder whose content has been customized in the WebForm.
10. The order in which events are raised: Load(Page) a Load(Master) a LoadCompete(page) i.e if we want to overwrite something already done in load event handler of Master then it should be coded in the LoadComplete event of the page.
11. Page_Load is the name of the method for the event handler for the Load event of Master. (it's not Master_Load).

Tracing in .NET -

Tracing helps to see the information of issues at the runtime of the application. By default tracing is disabled. Tracing has the following important features.

1. We can see the execution path of the page and application using the debug statement.
2. We can access and manipulate trace messages programmatically.
3. We can see the most recent tracing of the data.

Tracing can be done with the following 2 types -

1. Page Level When the trace output is displayed on the page and for the page-level tracing we need to set the property of tracing at the page level. <%@ Page trace=”true” Language=”C#” %>
2. In Application level tracing the information is stored for each request of the application. The default number of requests to store is 10. But if you want to increase the number of requests and discard the older request and display a recent request then you need to set the property in the web.config file. <trace enabled=”true”/>

Data Controls in ASP.NET -

The Controls having dataSource property are called Data controls in ASP.NET. ASP.NET allows a powerful feature of data binding, you can bind any server control to simple properties, collections, expressions and/or methods. When you use data binding, you have more flexibility when you use data from a database or other means. Data Bind controls are container controls. Controls->Child Control data Binding is binding controls to data from databases. With data binding we can bind a control to a particular column in a table from the database or we can bind the whole table to the data grid. Data binding provides a simple, convenient and powerful way to create a read/write link between the controls on a form and the data in their application. Data binding allows you to take the results of properties, collection, method calls, and database queries and integrate them with your ASP.NET code. You can combine data binding with Web control rendering to relieve much of the programming burden surrounding web control creation. You can also use data binding with ADO.NET and Web controls to populate control contents

from SQL select statements or stored procedures. Data Binding uses a special syntax <%# %> the <%#, which instructs ASP.NET to evaluate the expression. The difference between data binding tags are %> becomes apparent when the expression is evaluated. Expressions within the data binding tags are evaluated only when the DataBind method in the Page objects or Web Control is called. Data Bind Control can display data in connected and disconnected models.
Following are the data bind controls in ASP.NET-

- Repeater Control
- DataGrid control
- DataList control
- GridView Control
- DetailsView
- FormView
- DropDownList
- ListBox
- RadioButtonList
- CheckBoxList
- BulletList

Major events in global.asax-

The Global.asax file, which is derived from the HttpApplication class, maintains a pool of HttpApplication objects, and assigns them to applications as needed. The Global.asax file contains the following events.

- Application_Init
- Application_Disposed
- Application_Error
- Application_Start
- Application_End
- Application_BeginRequests

Use of CheckBox in .NET -

The CheckBox control is a very common control for HTML, unlike radio buttons it can select multiple items on a webpage. The CheckBox Control in ASP.NET has many properties and some of them are listed below-

Property	Description
AutoPostback	Specifies whether the form should be posted immediately after the checked property has changed or not. The default is false.
CausesValidation	Specifies if a page is validated when Button control is clicked.
Checked	Specifies whether the checkbox is checked or not.
InputAttributes	Attribute names and values used for the input element for the CheckBox control.
LabelAttributes	Attribute names and values used for the Label element for the CheckBox control.

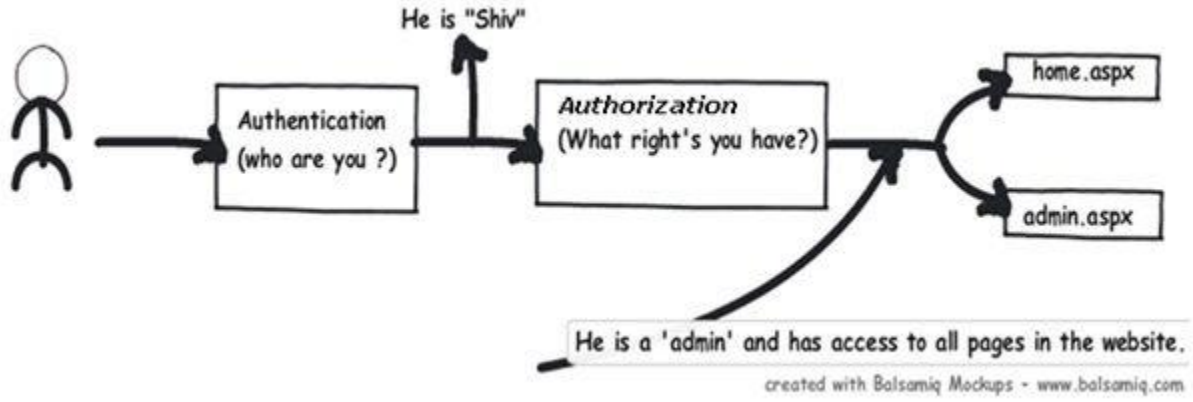
runat	Specifies that the control is a server control. Must be set to “server”
Text	The text next to the check box
TextAlign	On which side of the checkbox the text should appear (right or left)
ValidationGroup	Group of controls for which the Checkbox control causes validation when it posts back to the server
OnCheckedChanged	The name of the function to be executed when the checked property has changed.

Authentication and Authorization in ASP.NET -

Authentication - prove genuineness

Authorization - process of granting approval or permission on resources.

In ASP.NET authentication means to identify the user or in other words it’s nothing but validating that he exists in your database and he is the proper user. Authorization means he has access to a particular resource on the IIS website. A resource can be an ASP.NET web page, media files(MP4, GIF, JPEG etc.) compressed file (ZIP, RAR) etc.



Types of Authentication and Authorization in ASP.NET -

1. Windows Authentication -
   In this methodology ASP.NET web pages will use local windows users and groups to authenticate and authorize resources
2. Forms Authentication -
   This is a cookie based authentication where username and password are stored on client machines as cookie files or the data are sent through URL for every request. Form-based authentication presents the user with an HTML-based web page that prompts the user for credentials.
3. Passport Authentication -
   Passport authentication is based on the passport website provided by Microsoft. So when a user logins with credentials it will be reached to the passport website (i.e. hotmail,devhook,windows etc) where authentication will happen. If Authentication is successful it will return a token to your website.

Anonymous Access -

If you do not want any kind of authentication then you will go for Anonymous access.

In 'web.config' file set the authentication mode to 'Windows' as shown in the below code snippets

<authentication mode="windows">

We also need to ensure that all users are denied except authorized users. The below code snippet inside the authorizing tag that all users are denied. '?' indicates any unknown user.

```
<authorization>
        <deny users="?">
</authorization>
```

HTML server controls in ASP.NET -

The Microsoft.NET framework provides a rich set of server-side controls for developing web applications. You can add these controls to WebForms pages just as you add Windows controls to a form. Server-side controls are often called server controls or Web Forms control. There are four types of server controls; HTML server controls. Web server controls, validation control and user controls.

- HTML Server Controls -

  HTML developers must be familiar with old HTML controls, which they use to write GUI applications in HTML. These controls are the same HTML controls; you can run these controls on the server by defining the runat="server" attribute. These control names start with html.

Controls	Description
HtmlForm	Create an HTML form control, used as a placeholder of other controls.
HtmlInputText	Creates an input text box control used to get input from user
HtmlTextArea	Creates multiline text box control
HtmlAnchor	Creates a Web navigation
HtmlButton	Creates a button control
HtmlImage	Creates an image control, which is used to display an image.
HtmlInputCheckBox	Creates a checkbox control
HtmlInputRadioButton	Creates a radio button control
HtmlTable	Creates a table control
HtmlTableRow	Creates a row within a table
HtmlTableCell	Creates a cell within a row.

  - Web Server Controls
  - Validation Controls

- User Controls

Web API in ASP.NET -

It is a framework provided by Microsoft for writing HTTP services. There are many frameworks available to build HTTP based services. They follow a common guideline of international standardization but with different flavors. For example, all frameworks must adhere to these status codes.

- 1xx - Information Message
- 2xx - Successful
- 3xx - Redirection
- 4xx - Client Error
- 5xx - Server Error

Features-

- It is lightweight and thus good for small devices such as tables, smart phones.
- No tedious & extensive configuration like WCF REST is required.
- MediaTypeFormatter makes it easy to configure your APIs response type in a single line (JSON, XML and so on).
- IIS Hosting dependency is no more and it can be hosted in application too.
- Easy and simple control with HTTP features such as caching, Versioning, request/response headers and its various content formats.
- It supports content-negotiation (deciding the best response data format that client can accept).

ASP.NET Page life Cycle -

When a page is requested by the user from the browser, the request goes through a series of steps and many things happen in the background to produce the output or send the response back to the client.

The periods between the request and response of a page is called the “page life Cycle”

- Request - Start of the life cycle (sent by the user)
- Response - End of the life cycle (sent by the server)

There are four stages that occur during the Page life Cycle before the HTML Response is returned to the client.

1. Initialization

   During this stage the IsPostBack property is set. The page determines whether the request is a Postback (old request) or if this is the first time the page is being processed (new request).

   Controls on the page are available and each control’s uniqueID property is set. Now if the current request is a postback then the data has not been loaded and the value of the controls have not yet been restored from the view state.
2. Loading

   At this stage if the request is a Postback then it loads the data from the view state.
3. Rendering

   Before rendering, the View State is saved for the page and its controls. During this phase, the page calls the render method for each control, providing a text writer that writes its output to the OutputStream of the Page’s Response property.
4. Unloading -

Unload is called after the page has been fully rendered, sent to the client and is ready to be discarded. At this point also the page properties such as Response and Request are unloaded.

ASP.NET Page life Cycle events -

We have many events in ASP.NET page life cycle lets see some most important events:

- Page Request - When ASP.NET gets a page request, it decides whether to parse and compile the page or there would be a cached version of the page; accordingly the response is sent.
- Starting of Page life Cycle At this stage, the Request and Response objects are set. If the request is an old request or post back, the IsPostBack property of the page is set to true. The UICulture property of the page is also set.
- Page initialization At this stage, the controls on the page are assigned a unique ID by setting the UniqueID property and themes are applied. For a new request postback data is loaded and the control properties are restored to the view-state values.
- Page load At this stage, control properties are set using the view state and control state values.
- Validation Validate method of the validation control is called and if it runs successfully, the IsValid property of the page is set to true.
- Postback event handling If the request is a postback (old request), the related event handler is called.
- Page rendering At this stage, view state for the page and all controls are saved. The page calls the Render method for each control and the output of rendering is written to the OutputStream class of the Page's Response property.
- Unload The rendered page is sent to the client and page properties, such as Response and Request are unloaded and all cleanup done..

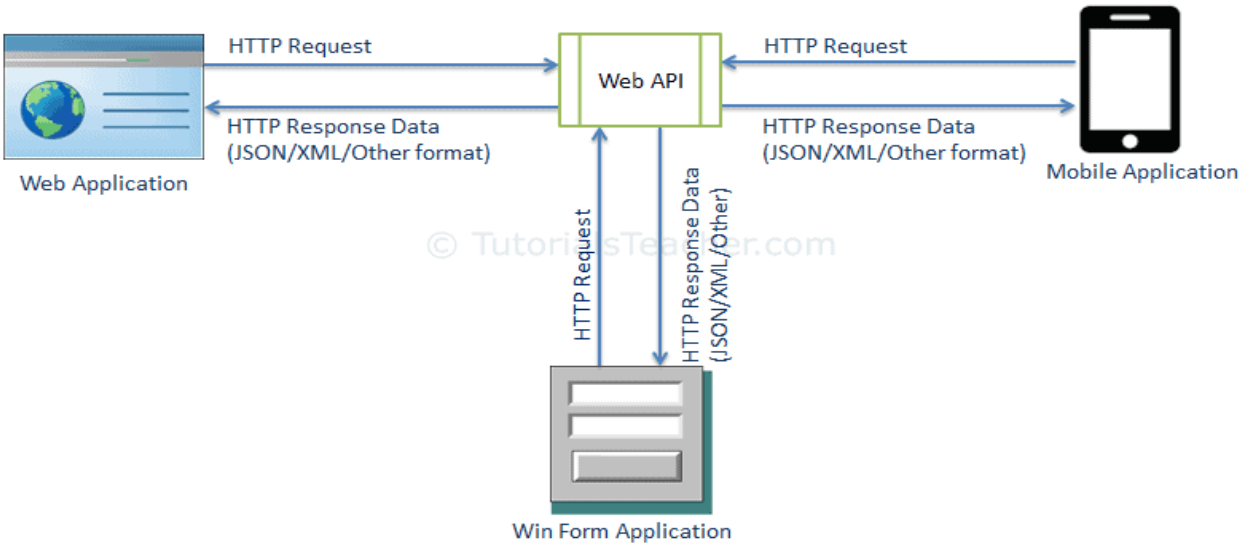
ASP.NET Page life Cycle Events Following are the page life cycle events -

- PreInit - preInit is the first event in the page life cycle. It checks the IsPostBack property and determines whether the page is a postback. It sets the themes and master pages, creates dynamic controls and gets and sets profile property values. This event can be handled by overloading the OnPreInit method or creating Page_PreInit handler.
- Init - Init event initializes the control property and the control tree is built. This event can be handled by overloading the OnInit method or creating a Page_Init handler.
- InitComplete - InitComplete event allows tracking of view state. All the controls turn on view-state tracking.
- LoadViewState - LoadViewState event allows loading view state information into the controls.
- LoadPostData - During this phase, the contents of all the input fields defined with the <form> tag are processed.
- PreLoad - PreLoad occurs before the postback data is loaded in the controls. This event can be handled by overloading the OnPreLoad method or creating a page_PreLoad handler.
- Load - the Load event is raised for the page first and then recursively for all child controls. The controls in the control tree are created. This event can be handled by overloading the OnLoad method or creating a page_load handler.

- LoadComplete - The loading process is completed, control event handlers are run and page validation takes place. This event can be handled by overloading the OnLoadComplete method or creating a page_LoadComplete handler.
- PreRender - The PreRender event occurs just before the output is rendered. By handling this event, pages and controls can perform any updates before the output is rendered.
- PrerenderComplete - As the Prerender event is recursively fired for all child controls, this event ensures the completion of the pre-rendering phase.
- SaveStateComplete - State of control on the page is saved. Personalization, control state and view state information is saved. The HTML markup is generated. This stage can be handled by overriding the Render method or creating a page_Render handler.
- UnLoad - The UnLoad phase is the last phase of the page life cycle. It raises the UnLoad event for all controls recursively and lastly for the page itself. Final cleanup is done and all resources and references, such as database connections, are freed. This event can be handled by modifying the OnUnLoad method or creating a page_UnLoad handler.

ASP.NET Web API -

The ASP.NET Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms such as web, windows, mobile etc. It works more or less the same way as ASP.NET MVC web application except that it sends data as a response instead of html view. It is like a web service or WCF service but the exception is that it only supports HTTP protocol.



ASP.NET Web API Characteristics -

1. ASP.NET Web API is an ideal platform for building RESTful services.
2. ASP.NET Web API is built on top of ASP.NET and supports ASP.NET request/response pipeline.
3. ASP.NET Web API maps HTTP verbs to method names.
4. ASP.NET Web API supports different formats of response data. Built-in support for JSON, XML,BSON format.

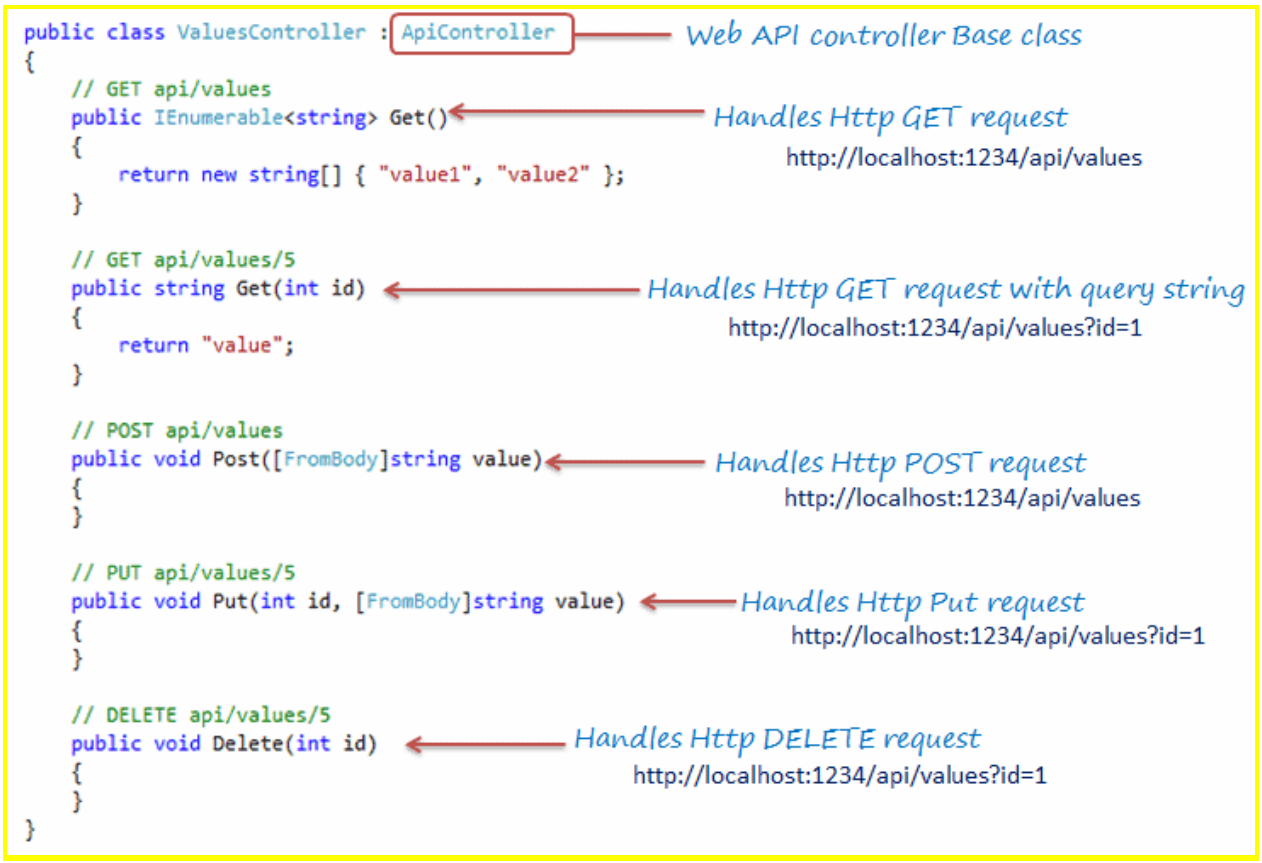
5. ASP.NET Web API can be hosted in IIS, Self-hosted or other web server that supports .NET 4.0+.
6. ASP.NET Web API framework includes new HttpClient to communicate with Web API server. HttpClient can be used in ASP.NET MVC server side, Windows Form application, Console application or other apps.

Web API	**WCF**
Open source and ships with .NET framework.	Ships with .NET framework.
Supports only HTTP protocol.	Supports HTTP, TCP, UDP and custom transport protocol.
Maps http verbs to methods	Uses attributes based programming model.
Uses routing and controller concept similar to ASP.NET MVC.	Uses Service, Operation and Data contracts.
Does not support Reliable Messaging and transaction.	Supports Reliable Messaging and Transactions.
Web API can be configured using HttpConfiguration class but not in web.config.	Uses web.config and attributes to configure a service.
Ideal for building RESTful services.	Supports RESTful services but with limitations.

Web API Controllers -

Web API Controller is similar to ASP.NET MVC Controller. It handles incoming HTTP requests and send response back to the caller.

Web API Controller is a class which can be created under the Controllers folder or any other folder under your project's root folder. The name of a controller class must end with "Controller" and it must be derived from System.Http.ApiController class. All the public methods of the controller are called action methods.



As you can see in the above example, ValuesController class is derived from APIController and includes multiple action methods whose names match with HTTP verbs like Get, Post, Put and Delete.

Based on the incoming request URL and HTTP verb (GET/POST/PUT/PATCH/DELETE), Web API decides which Web API controller and action method to execute e.g. Get() method will handle HTTP GET request, Post() method will handle HTTP POST request, Put() method will handle HTTP PUT request and Delete() method will handle HTTP DELETE request for the above Web API.

If you want to write methods that do not start with an HTTP verb then you can apply the appropriate http verb attribute on the method such as HttpGet, HttpPost, HttpPut etc. same as MVC controller.

```
Using System;
Using System.ColeIctions.Generic;
Using System.Linq;
Using System.Net;
Using System.Net.Http;
Using System.Web.Http;
Namespace MyWebAPI.Controllers
{
        Public class ValuesController : ApiController
        {
        [HttpGet]
        Public IEnumerable<string> values()
```

```
        {
            Return new string[] {“value1”, “value2”};
        }
        [Httpget]
        Public string value(int id)
        {
            Return “value”;
        }
        [HttpPost]
        Public void SaveNewValue([FromBody]string value)
        {}
        [HttpPut]
        Public void Updatevalue(int id, [FromBody]string value)
        {}
        [HttpDelete]
        Public void removevalue(int id)
        {}
        }
}
```

Web API Controller Characteristics -

1. It must be derived from System.Web.Http.ApiController class.
2. It can be created under any folder in the project’s root folder. However, it is recommended to create controller classes in the Controllers folder as per the convention.
3. Action method name can be the same as HTTP verb name or it can start with HTTP verb with any suffix(case in-sensitive) or you can apply Http verb attributes to method.
4. Return type of an action method can be any primitive or complex type.

HTTP Method	**Possible Web API Action Method Name**	**Usage**
GET	get() get() GET() GetAllStudent() *any name starting with Get*	Retrieves data
POST	Post() post() POST() PostNewStudent() *any name starting with Post*	Insert new record
PUT	Put() put() PUT() PutStudent() *any name starting with Put*	Updates existing record
PATCH	Patch()	Updates records partially

	patch() PATCH() PatchStudent() *any name starting with Patch*	
DELETE	Delete() delete() DELETE() DeleteStudent() *any name starting with Delete*	Deletes record

Web API Controller	**MVC Controller**
Derives from System.Web.Http.ApiController class	Derives from System.Web.MVC.Controller class
Method name must be start with Http verbs otherwise apply http verbs attribute	Must apply appropriate Http verbs attribute
Specialized in returning data	Specialized in rendering view
Return data automatically formatted based on Accept-Type header attribute to json or xml	Returns ActionResult or any derived type.
Requires .NET 4.0 above	Requires .NET 3.5 or above

Configure Web API -

Web API supports code based configuration. It cannot be configured in web.config file. We can config Web API to customize the behaviour of Web API hosting infrastructure and components such as routes, formatters, filters, Dependencyresolver, Messagehandlers, ParameterBindingRules, properties, services etc.

Web API configuration process starts when the application starts. It calls GlobalConfiguration.Config(WebApiConfig.Register) in the Application_Start method. The Configure() method requires the callback method where Web API has been configured in code. By default this is the static WebApiConfig.Register() method.

Global.asax

```
Public class WebAPIApplication : System.Web.HttpApplication
{
        Protected void Application_Start()
        {
                GlobalConfiguration.Configure(WebApiConfig.Register);
                //other configuration
        }
}
```

WebApiConfig

```
Public static class WebApiConfig
```

```
{
        Public static void Register(httpConfiguration config)
        {
                config.MapHttpAttributeRoutes();
                config.Routes.MapHttpRoute(
                        Name : “DefaultApi”;
                        routeTemplate : “api/{controller}/{id}”;
                        Defaults : new {id = RouteParameter.Optional}
                );
                //configure additional webapi settings here…
        }
}
```

Property	Description
DependencyResolver	Gets or sets the dependency resolver for dependency injection.
Filters	Gets or sets the filters.
Formatters	Gets or sets the media-type formatters.
IncludeErrorDetailPolicy	Gets or sets value indicating whether error details should be included in error messages.
MessageHandlers	Gets or sets the message handlers.
ParameterBindingRules	Gets the collection of rules for how parameters should be bound.
Properties	Gets the properties associated with this Web API instance.
Routes	Gets the collection of routes configured for the Web API.
Services	Gets the Web services.

Web API Routing -

Web API routing is similar to ASP.NET MVC Routing. It routes an incoming HTTP request to a particular action method on a Web API controller.

Web API supports two types of routing -

1. Convention-based Routing
2. Attribute Routing

Convention-based Routing -

In the convention-based routing, Web API uses route templates to determine which controller and action method to execute. At least one route template must be added into route table in order to handle various HTTP requests.

When we created Web API project using WebAPI template in the Create Web API Project section, it also added WebApiConfig class in the App_Start folder with default route as shown below.

WebApiConfig with Default Route

```
Public static class WebApiConfig
{
        Public static void Register(HttpConfiguration config)
        {
                //Enable attribute routing
                config.MapHttpAttributeRoutes();
                //Add default route using convention-based routing
                config.Routes.MapHttpRoute(
                        name:"DefaultApi",
                        routeTemplate:"api/{controller}/{id}",
                        Defaults: new {id=RouteParameter.Optional}
                        );
        }
}
```

In the above WebConfig.Register() method, config.MapHttpAttributeRoute() enables attribute routing which we will learn later in this section. The config.Routes is a route table or route collection of type HttpRouteCollection. The "DefaultApi" route is added in the route table using MapHttpRoute() extension method. The MapHttpRoute() extension method internally creates a new instance of IHttpRoute and adds it to an HttpRouteCollection. However, you can create a new route and add it into a collection manually as shown below.

Add Default Route -

```
Public static class WebApiConfig
{
        Public static void Register(HttpConfiguration config)
        {
                config.MapHttpAttributeRoutes();
                //define route
                IHttpRoute defaultRoute=config.Routes.CreateRoute("api/{controller}/{id}",
                        new {id=RouteParameter.Optional}, null);
                //Add route
                config.Routes.Add("DefaultApi", defaultRoute);
        }
}
```

The following table lists parameters of MapHttpRoute() method.

Parameter	Description
name	Name of the route
routetemplate	URL pattern of the route
defaults	An object parameter that includes default route values
constraints	Regex expression to specify characteristic of route values
handler	The handler to which the request will be dispatched.

Now, let's see how Web API handles an incoming http request and sends the response.

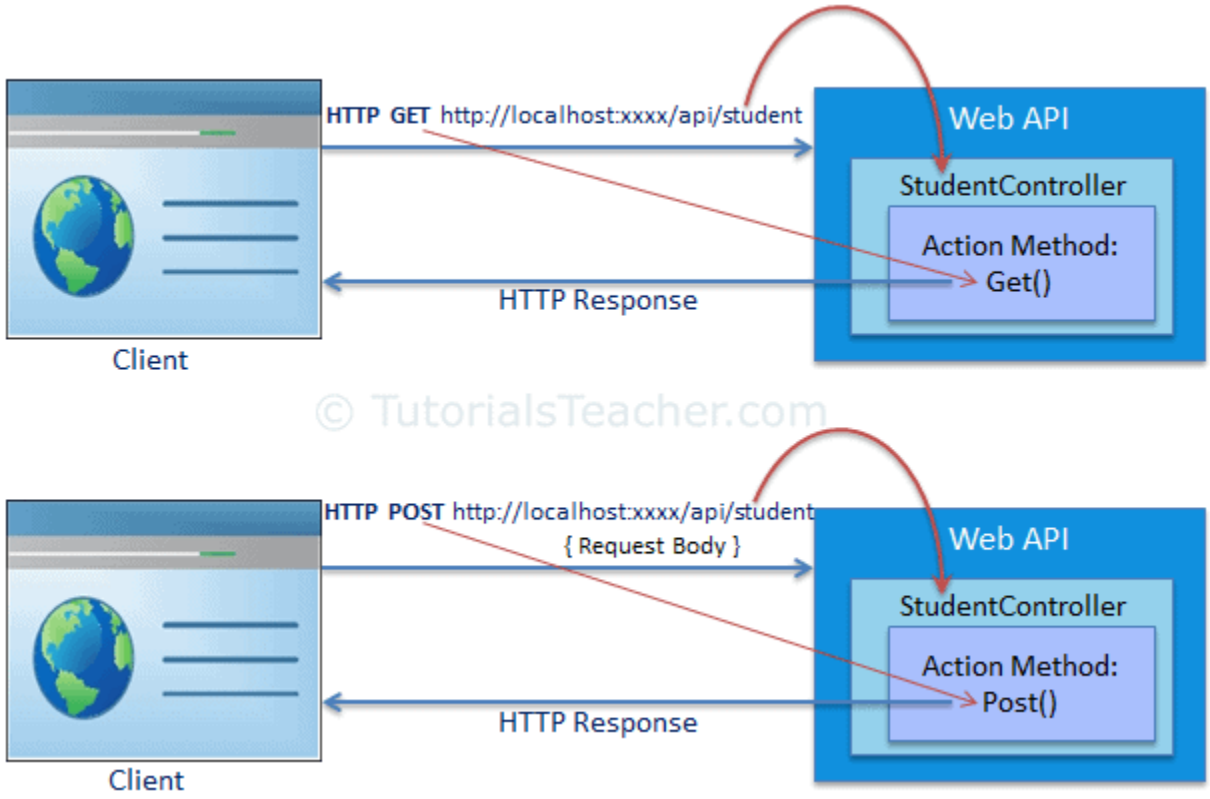
The following is a sample HTTP GET request.

Sample HTTP GET Request -

Get http://localhost:1234/api/values/HTTP/1.1
User-Agent Fiddler
Host: localhost: 60464
Content-Type: application/json

Considering the DefaultApi route configured in the above WebApiConfig class, the above request will execute Get() action method of the ValuesController because HTTP method is a GET and URL is http://localhost:1234/api/values which matches the DefaultApi's route template /api/{controller}/{id} where value of {controller} will be ValuesController. Default route has specified id as an optional parameter so if an id is not present in the url then {id} will be ignored. The request's HTTP method is GET so it will execute Get() action method of ValueController.

If Web API framework does not find matched routes for an incoming request then it will send 404 error response.



Configure Multiple Routes -

We configured single route above. However, you can configure multiple routes in the Web API using HttpConfiguration object. The following example demonstrates configuring multiple routes.

Multiple Routes -

```
Public static class WebApiConfig
{
        Public static void Register(HttpConfiguration config)
        {
        config.MapHttpAttributeRoutes();
        //school route
        config.Routes.MapHttpRoute(
                name:"School",
```

```
                routeTemplate:"api/myschool/{id}",
                Defaults: new {controller="school", id=Routeparameter.Optional}
                Constraints: new {id="/d+"}
                );
        //default route
        config.Routes.MapHttpRoute(
                name:"DefaultApi",
                routeTemplate:"api/{controller}/{id}",
                Defaults: new {id=RouteParameter.Optional}
                )
        }
}
```

In the above example, the School route is configured before the DefaultApi route. So any incoming request will be matched with the school route and if the incoming request url does not match with it then only it will be matched with the DefaultApi route. For example, the request url is http://localhost:1234/api/myschool is matched with the School route template, so it will be handled by SchoolController.

Note - The reason to use api in the route template is just to avoid confusion between MVC controller and Web API controller. You can use any pattern based on your app architecture.

Attribute Routing -

Attribute routing is supported in Web API 2. As the name implies, attribute routing uses [Route()] attribute to define routes. The route attribute can be applied on any controller or action method.

In order to use attribute routing with Web API, it must be enabled in WebApiConfig by calling config.MapHttpAttributeRoutes() method.

Consider the following example of attribute routing.

Example: Attribute Routing-

```
Public class StudentController: ApiController
{
        [Route("api/student/names")]
        Public IEnumerable<string> get()
        {
                Return new string[] {"student1", "student2"}
        }
}
```

In the above example, the Route attribute defines new route "api/student/names" which will be handled by the Get() action method of StudentController. Thus, an HTTP GET request http://localhost:1234/api/student/names will return list of student names.

Parameter Binding in ASP.NET Web API -

Action methods in Web API controllers can have one or more parameters of different types. It can be either primitive type or complex type. Web API binds action method parameters with the URL's query string or with the request body depending on the parameter type.

By default, if the parameter type is of .NET primitive types such as int, bool, double, string, GUID, DateTime, decimal, or any other type that can be converted from string type, then it sets the value of a parameter from the query string. And if the parameter type is the complex type, then Web API tries to get the value from the request body by default.

HTTP Method	Query String	Request Body
GET	Primitive Type, Complex Type	NA
POST	Primitive Type	Complex Type
PUT	Primitive Type	Complex Type
PATCH	Primitive Type	Complex type
DELETE	Primitive Type, Complex Type	NA

Get Action Method with Primitive Parameter -

Consider the following example of the GET action method that includes a single primitive type parameter.

Example: Primitive Parameter Binding

```
Public class StudentController : ApiController
{
        Public Student Get(int id)
        {}
}
```

As you can see, the above HTTP GET action method includes the id parameter of the int type. So Web API will try to extract the value of id from the query string of the requested URL, convert it into int and assign it to the id parameter of the GET action method. For example, if an HTTP request is http://localhost/api/student?id=1 then the value of the id parameter will be 1.

The followings are valid HTTP GET requests for the above action method.

http://localhost/api/student?id=1

http://localhost/api/student?ID=1

Note- Query string parameter name and action method parameter name must be the same (case-insensitive). If names do not match, then the values of the parameters will not be set. The order of the parameters can be different.

Multiple Primitive Parameters -

Consider the following example of the GET action method with multiple primitive parameters.

Example: Multiple Parameters Binding

```
Public class StudentController : ApiController
{
        Public Student get(int id, string name)
        {}
}
```

As you can see above, an HTTP GET method includes multiple primitive type parameters. So, Web API will try to extract the values from the query string of the requested URL. For example, If an HTTP request is http://localhost/api/student?id=1&name=steve, then the value of the is parameter will be 1, and the name parameter will be “steve”.

Followings are valid HTTP GET Requests for the above action method.

http://localhost/api/student?id=1&name=steve

http://localhost/api/student?ID-1&NAME=steve

http://localhost/api/student?name=steve&id=1

Note- Query string parameters names must match with the name of an action method parameter. However, they can be in a different order.

Post Action Method with Primitive Parameter -

An HTTP POST request is used to create a new resource. It can include request data into the HTTP request body and also in the query string.

Consider the following Post action method.

Example: Post Method with Primitive Parameter

```
Public class StudentController : ApiController
{
        Public Student Post(id id, string name)
        {}
}
```

As you can see above, the Post() action method includes primitive type parameters id and name. So by default, Web API will get values from the query string. For example, if an HTTP POST request is http://localhost/api/student?id=1&name=steve, then the value of the id parameter will be 1 and the name parameter will be “steve” in the above Post() method.

Now, consider the following Post() method with the complex type parameter.

Example: Post Method with Complex Type Parameter

```
Public class student
{
        Public int Id {get; set;}
        Public string Name {get; set;}
}
Public class StudentController : ApiController
{
        Public Student Post(Student stud)
        {}
}
```

The above post() method includes the Student type parameter. So, as a default rule, Web API will try to get the values of the stud parameter from the HTTP request body.

Web API will extract the JSON object from the HTTP request body above, and convert it into a Student object automatically because the names of JSON object’s properties match with the name of the Student class properties. (case-insensitive).

POST Method with Mixed Parameters -

The HTTP Post action methods can include primitive and complex type parameters. Consider the following example.

Example - Post Method with Primitive and Complex Type Parameters

```
Public class student
{
        Public int ID {get; set;}
        Public string Name {get; set;}
}
Public class StudentController : ApiController
{
        Public Student Post(int age, Student student)
        {}
```

```
}
```
The above Post method includes both primitive and complex type parameters. So, by default, Web API will get the age parameter from query string and student parameter from the request body.

Note- Post action method cannot include multiple complex type parameters because, at most, one parameter is allowed to be read from the request body.

Parameter binding for Put and Patch method will be the same as the POST method in Web API.

[FromUri] and [FormBody]

You have seen that by default ASP.NET Web API gets the value of a primitive parameter from the query string and complex type parameter from the request body. But, what if we want to change this default behaviour?

Use [FromUri] attribute to force Web API to get the value of complex type from the query string and [FromBody] attribute to get the value of primitive type from the request body, opposite to the default rules.

Example : FromUri

```
Public class StudentController:ApiController
{
        Public Student Get([FromUri] Student stud)
        {}
}
```

In the above example,the Get() method includes a complex type parameter with the [FromUri] attribute. So, Web API will try to get the value of the Student type parameter from the query string. For example, if an HTTP GET request http://localhost:xxxx/api/student?id=1&name=steve then Web API will create an object the Student type and set its id and name property values to the value of id and name query string parameter.

Note - the name of the complex types properties and the query string parameters must match.

In the same way, consider the following example of post() method.

We have applied the [FromUri] attribute with the Student parameter. By default, Web API extracts the value of the complex type from the request body, but here, we have applied the [FromUri] attribute. So now, Web API will extract the value of the Student properties from the query string instead of the request body.

In the same way, apply the [FromBody] attribute to get the value of primitive data type from the request body instead of a query string, as shown below

Example: FromBody

```
Public class StudentController : ApiController
{
        Public Student Post([FromBody] Student stud)
        {}
}
```

Note - The [FromBody] attribute can be applied on only one primitive parameter of an action method. It cannot be applied to multiple primitive parameters of the same action method.

Action Method Return Type -

The Web API action method can have following return types.

1. Void -

It's not necessary that all action methods must return something. It can have void return type.

For example, consider the following Delete action method that just deletes the student from the data source and returns nothing.

Void Return Type-

```
Public class StudentController : ApiController
{
        Public void Delete(int id)
        {
                DeleteStudentFromDB(id);
        }
}
```

As you can see above Delete action method returns void. It will send 204 "No Content" status code as a response when you send HTTP DELETE request as shown below.

2. Primitive type or Complex type -

   An action method can return primitive or custom complex types as other normal methods.

   Consider the following Get action methods.

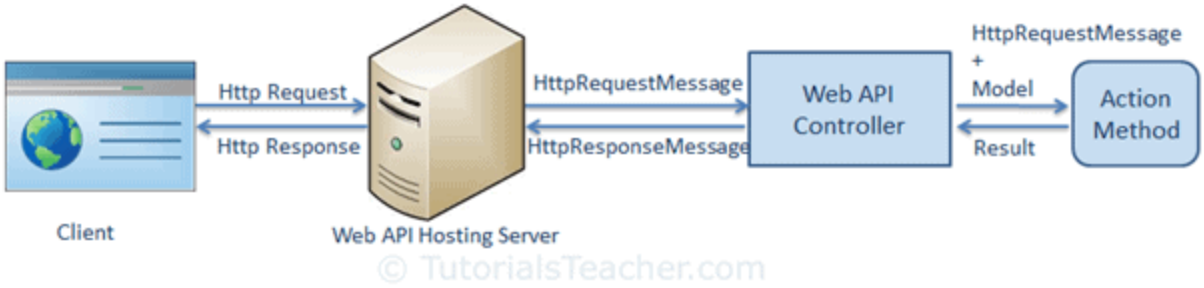
Primitive or Complex Return Type-

```
Public class Student
{
        Public int Id{ get; set; }
        Public string Name{ get; set; }
}
Public class StudentController : ApiController
{
        Public int GetId(string name)
        {
                Int id= GetStudentId(name);
                Return id;
        }
        Public Student GetStudent(int id)
        {
                Var student=GetStudentFromDB(id);
                Return student;
        }
}
```

As you can see above, GetId action method return an integer and GetStudent action method returns a Student type.

3. HttpResponseMessage -

   Web API controller always returns an object of HTTPResponseMessage to the hosting infrastructure. The following figure illustrates the overall Web API request/response pipeline.



As you can see in the above figure, the Web API controller returns HttpResponseMessage object. You can also create and return an object of HttpResponseMessage directly from an action method.

The advantage of sending HttpResponseMessage from an action method is that you can configure a response your way. You can set the status code, content or error message (if any) as per your requirement.

Return HttpResponseMessage-

```
Public HttpResponseMessage Get(int id)
{
        Student stud=GetStudentFromDB(id);
        if(stud==null)
        {
                Return Request.CreateResponse(HttpStatusCode.NotFound, id);
        }
        Return Request.CreateResponse(HttpStatusCode.OK, stud);
}
```

In the above action method, if there is no student with specified id in the DB then it will return HTTP 404 Not Found status code, otherwise it will return 200 OK status with student data.

4. IHttpActionResult -

The IHttpActionResult was introduced in Web API 2 (.NET 4.5). An action method in Web API 2 can return an implementation of IHttpActionResult class which is more or less similar to ActionResult class in ASP.NET MVC.

You can create your own class that implements IHttpActionResult or use various methods of ApiController class that returns an object that implement the IHttpActionResult.

Return IHttpActionResult Type using OK() and NotFound() methods -

```
Public IHttpActionResult Get(int id)
{
        Student stud=GetStudentFromDB(id);
        If (stud==null)
        {
                Return NotFound();
        }
        Return Ok(stud);
}
```

In the above example, if a student with specified id does not exist in the database then it will return a response with the status code 404 otherwise it sends student data with

status code 200 as a response. As you can see, we don't have to write much code because the NotFound() and Ok() method does it all for us.
The following table lists all the methods of the ApiController class that returns an object of a class that implements the IHttpActionResult interface.

ApiController Method	Description
BadRequest()	Creates a BadRequestResult object with status code 400.
Conflict()	Creates a ConflictResult with status code 409.
Content()	Creates a NegotiatedContentResult with the specified status code and data.
Created()	Creates a CreatedNegotiatedContentResult with status code 201 created.
CreatedAtRoute()	Creates a CreatedAtRouteNegotiatedContentResult with status code 201 created.
InternalServerError()	Creates an InternalServerErrorResult with status code 500 Internal server error.
NotFound()	Creates a NotFoundResult with status code 404
Ok()	Creates an OkResult with status code 200.
Redirect()	Creates a RedirectResult with status code 302.
RedirectToRoute()	Creates a RedirectToRouteResult with status code 302.
ResponseMessage()	Creates a ResponseMessageResult with the specified HttpResponseMessage.
StatusCode()	Creates a StatusCodeResult with specified http status code.
Unauthorized()	Creates an UnauthorizedResult with status code 401.

Web API Request/Response Data Formats -

Media type -

Media type (aka MIME type) specifies the format of the data as type/subtype e.g. text/html, text/xml, application/json, image/jpeg etc.
In HTTP request, MIME type is specified in the request header using Accept and Content-Type attribute. The Accept header attribute specifies the format of response data which the client expects and the Content-Type header attribute specifies the format of the data in the request body.
For example, if a client wants response data in JSON format then it will end following the GET HTTP request with the accepted header to the Web API.

HTTP Get Request -

```
GET http://localhost:60464/api/student HTTP/1.1
User-Agent: Fiddler
Host: localhost:1234
Accept: application/json
```

In the same way, if a client includes JSON data in the request body to send it to the receiver then it will send the following HTTP request with Content-Type header with JSON data in the body.

HTTP POST Request-

```
POST http://localhost:60464/api/student?age=15 HTTP/1.1
user-Agent: Fiddler
Host: localhost:60464
Content-Type: application/json
Content-Length: 13
{
        Id:1,
        name:'Steve'
}
```

Web API converts request data into CLR objects and also serializes CLR objects into response data based on Accept and Content-type headers. Web API includes built-in support for JSON, XML, BSON, and form-urlencoded data. It means it automatically converts request/response data into these formats OOB (out-of box).

Post Action Method -

```
Public class Student
{
        Public int Id{ get; set; }
        Public string Name{ get; set; }
}
Public class StudentController : ApiController
{
        Public Student Post(Student student)
        {
                //save student into db
                Var insertedStudent=SaveStudent(student);
                Return insertedStudent;
        }
}
```

As you can see above, the Post() action method accepts Student type parameters, saves that student into DB and returns inserted student with generated id. The above Web API handles HTTP POST requests with JSON or XML data and parses it to a Student object based on Content-Type header value and the same way it converts insertedStudent object into JSON or XML based on Accept header value.

ASP.NET Web API: Media-Type Formatters-

Media type formatters are classes responsible for serializing request/response data that Web API can understand the request data format and send data in the format which client expects.
Web API includes the following built-in media type formatters.

Media type Formatter Class	MIME Type	Description
JsonMediaFormatter	application/json, text/json	Handles JSON format
XmlMediaTypeFormatter	application/xml, text/json	Handles XML format
FormatUrlEncodedmediaTypeFormatter	application/x-www-form-urlencoded	Handles HTML form URL-encoded data
JQueryMvcFormUrlEncodedFormatter	application/x-www-form-urlencoded	Handles model-bound HTML form URL-encoded data.

Retrieve Built-in Media Type Formatters -

As mentioned Web API includes above listed media type formatter classes by default. However, you can also add, remove or change the order of formatters.

The following example demonstrates the HTTP Get method that returns all built-in formatter classes.

Retrieve Built-in Formatters in C#

```
Public class FormattersController: ApiController
{
        Public IEnumerable<string> Get()
        {
                IList<string> formatters =new List<string>();
                foreach(var item in GlobalConfiguration.Configuration.Formatters)
                {
                        formatters.Add(item.ToString());
                }
                Returns formatters.AsEnumerable<string>();
        }
}
```

In the above example, GlobalConfiguration.Configuration.Formatters returns MediatypeFormatterCollection that includes all the formatter classes.

Alternatively, MediaTypeFormatterCollection class defines convenience properties that provide direct access to three of the four built-in media type formatters. The following example demonstrates retrieving media type formatters using MediaTypeFormatterCollection's properties.

Retrieve Built-in Formatters in C# -

```
Public class FormattersController : ApiController
{
        Public IEnumerable<string> Get()
        {
        IList<string> formatters =new List<string>();
        formatters.Add(GlobalConfiguration.Configuration.Formatters.JsonFormatter.GetType().FullName
        );
        formatters.Add(GlobalConfiguration.Formatters.XmlFormatter.GetType().FullName);
        formatter.Add(GlobalConfiguration.Formatters.FormUrlEncodedFormatter.GetType().FullName);
        Return formatters.AsEnuerable<string>();
```

```
        }
}
```

BSON Formatter -

Web API also supports BSON format. As the name suggests, BSON is binary JSON, it is a binary-encoded serialization of JSON-like documents. Currently there is very little support for BSON and no JavaScript implementation is available for clients running in browsers. This means that it is not possible to retrieve and automatically parse BSON data to javaScript objects.

Web API includes built-in formatter class BsonMediaTypeFormatter for BSON but it is disabled by default.

JSON Formatter -

Web API includes JsonMediaTypeFormatter class that handles JSON format. The JsonMediaTypeFormatter converts JSON data in an HTTP request into CLR objects and also converts CLR objects into JSON format that is embedded within HTTP response.

Internally, JsonMediaTypeFormatter uses a third party open source library called Json.NET to perform serialization.

Configure JSON Serialization -

JSON formatter can be configured in WebApiConfig class. The JsonMediaTypeFormatter class includes various properties and methods using which you can customize JSON serialization. For example, Web API writes JSON property names with PascalCase by default. To write JSON property names with camelCase, set the CamelCasePropertyNamesContractResolver on the serialization settings as shown below-

Customize JSON Serialization in C# -

```
Public static class WebApiConfig
{
        Public static void Register(HttpConfiguration config)
        {
                config.MapHttpAttributeRoutes();
                config.Routes.MapHttpRoute(name: “DefaultApi”,
                        RouteTemplate: “api/{controller}/{id}”,
                        Defaults: new {id=RouteParameter.Optional});
                        //Configure json formatter
                        JsonMediaTypeFormatter jsonFormatter=config.Formatters.JsonFormetter;
                        jsonFormatter.SerializerSettings.ContractResolver = new
                CamelCasePropertyNamesContractResolver();
        }
}
```

XML Formatter -

The XmlMediaFormatter class is responsible for serializing model objects into XML data. It uses System.Runtime.DataContractSerializer class to generate XML data.

Web API Filters -

Web API includes filters to add extra logic before or after the action method executes. Filters can be used to provide cross-cutting features such as logging, exception handling, performance measurement, authentication and authorization.

Filters are actually attributes that can be applied on the Web API controller or one or more action methods. Every filter attribute class must implement IFilter interface included in

System.Http.Filters namespace. However System.Web.Http.Filters includes other interfaces and classes that can be used to create filters for specific purposes.

Filter Type	Interface	Class	Description
Simple Filter	IFilter	-	Defines the methods that are used in a filter.
Action Filter	IActionFilter	ActionFilterAttribute	Used to add extra logic before or after action methods execute.
Authentication Filter	IAuthenticationFilter	-	Used to force users or clients to be authenticated before action methods execute.
Authorization Filter	IAuthorizationFilter	AuthorizationFilterAttribute	Used to restrict access to action methods to specific users or groups.
Exception Filter	IExceptionFilter	ExceptionFilterAttribute	Used to handle all unhandled exceptions in Web API.
Override Filter	IOverride Filter	-	Used to customize the behaviour of other filters for individual action methods.

As you can see, the above table includes class as well as interface for some of the filter types. Interfaces include methods that must be implemented in your custom attribute class whereas filter class has already implemented necessary interfaces and provides virtual methods, so that they can be overridden to add extra logic. For example, the ActionFilterAttribute class includes methods that can be overridden. We just need to override methods which we are interested in, whereas if you use IActionFilter attribute then you must implement all the methods.

Let's create a simple LogAttribute class for logging purpose to demonstrate action filters.

Web API Filter Class -

```
Public class logAttribute:ActionFilterAttribute
{
        Public LogAttribute()
        {}
        Public override void OnActionExecuting(HttpActionContext actionContext)
        {
                trace.WriteLine(string.format("Action method {0} executed at {1}",
actionContext6.ActionDescriptor.ActionName, DateTime.Now.ToShortDateString()),"Web API logs");
}
Public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)
{
        Trace.WriteLine(string.Format("Action Method {0} executed at {1}",
actionExecutedContext.ActionText.ActionDescriptor.ActionName, DateTime.Now.ToShortDateString()),
"Web API log")
```

}
}
In the above example, logAttribute is derived from ActionFilterAttribute class and overridden OnActionExecuting and OnActionExecuted methods to log in the trace listeners.

Web API CRUD Operations-

New Web API project and implement GET, POST, PUT and DELETE method for CRUD operation using Entity Framework.

1. Get Method -
   Action method that starts with a word “get” will handle HTTP GET request. We can either name it only Get or with any suffix. Let’s add our first Get action method and give it a name GetAllStudents because it will return all the students from the DB.
2. Post Method -
   The HTTP POST request is used to create a new record in the data source in the RESTful architecture.
3. Put Method -
   The HTTP PUT method is used to update existing record in the data source in the RESTful architecture.
4. Delete Method -
   The HTTP DELETE request is used to delete an existing record in the data source in the RESTful architecture.

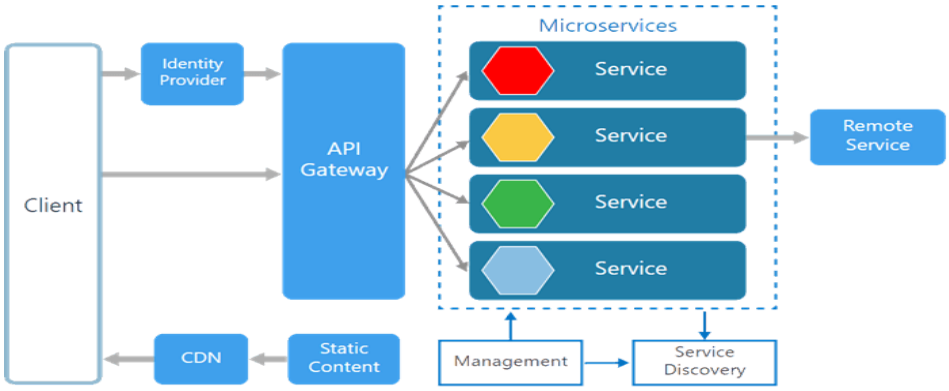
Web API Hosting -

1. IIS Hosting -
   Web API can be hosted under IIS, in the same way as a web application. A Web API is created with ASP.NET MVC project by default. So, when you host your MVC web application under IIS it will also host Web API that uses the same base address.
2. Self Hosting -
   You can host a Web API as a separate process than ASP.NET. It means you can host a Web API in console application or windows service or OWIN or any other process that is managed by .NET framework.
   You need to do the following steps in order to self-host a Web API.
   1. Use HttpConfiguration to configure a Web API.
   2. Create HttpServer and start listening to incoming http requests.

Microservices using ASP.NET core -

Microservices is more about applying a certain number of principles and architectural patterns as architecture. Each microservice lives independently., but on the other hand, also all rely on each other. All microservices in a project get deployed in production at their own pace, on premise on the cloud, independently, living side by side.

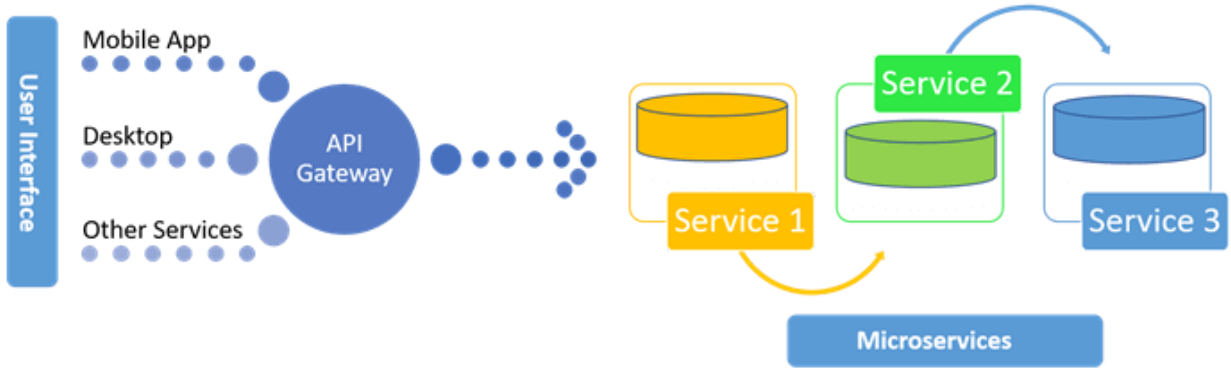
Microservice Architecture -



There are various components in a microservices architecture apart from microservices themselves.

1. Management - Maintains the nodes for the service.
2. Identity Provider- Manages the identity information and provides authentication services within a distributed network.
3. Service Discovery - Keeps track of services and service addresses and endpoints.
4. API Gateway - Services as client's entry point. Single point of contact from the client which in turn returns responses from underlying microservices and sometimes an aggregated response from multiple underlying microservices.
5. CDN - A content delivery network to serve static resources for e.g. pages and web content in a distributed network.
6. Static Content - The static resources like pages and web content.

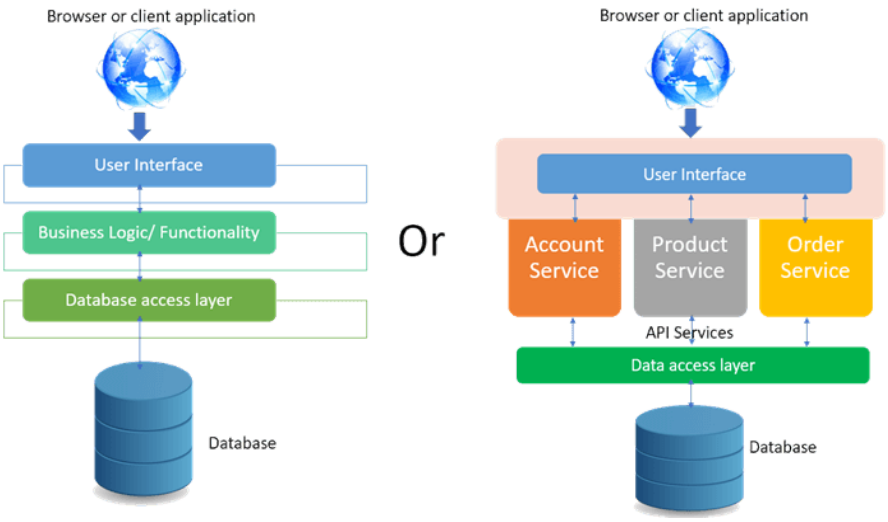
Microservices are deployed independently with their own database per service so the underlying microservices look as shown in the following picture.



Monolithic vs Microservices Architecture -

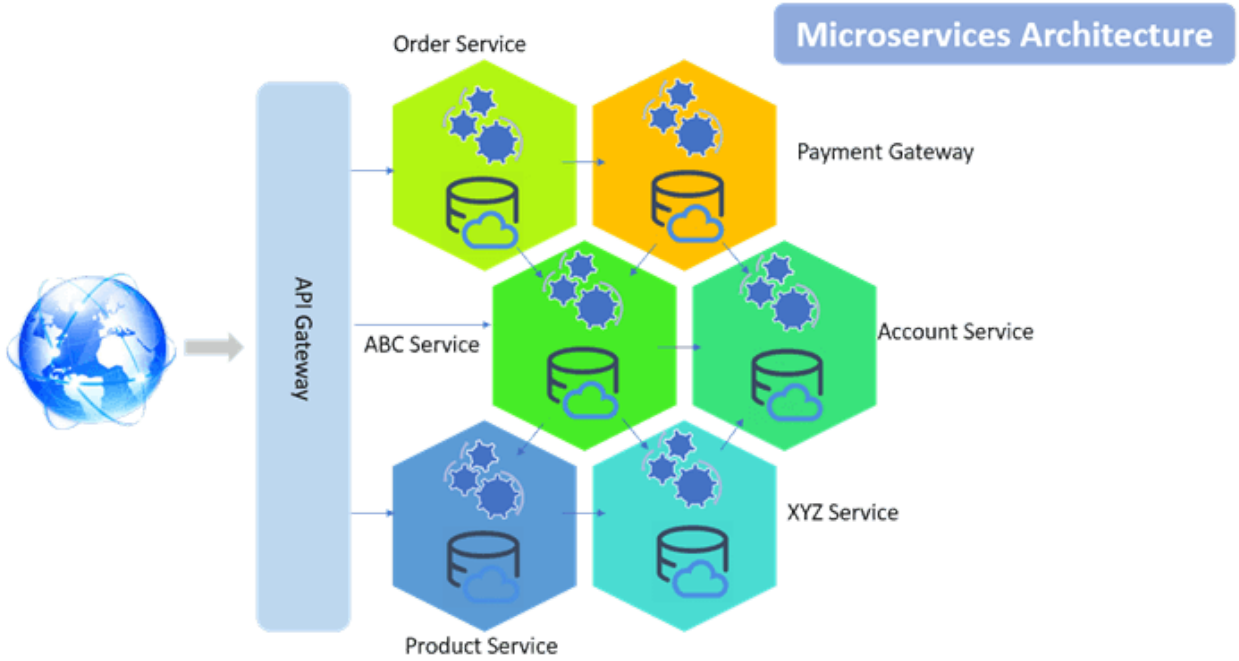
Monolithic applications are more of a single complete package having all the related needed components and services encapsulated in one package.

Following is the diagrammatic representation of monolithic architecture being packaged completely or being service based.



Microservice is an approach to create small services each running in their own space and can communicate via messaging. These are independent services directly calling their own database.

Following is the diagrammatic representation of microservices architecture.



In monolithic architecture, the database remains the same for all the functionalities even if an approach of service-oriented architecture is followed, whereas in microservices each service will have its own database.

Docker containers and Docker Installation -

Containers like Docker and others slice the operating system resources, for e.g. the network stack, processes namespace, file system hierarchy and the storage stack. Dockers are more like virtualizing the operating system.

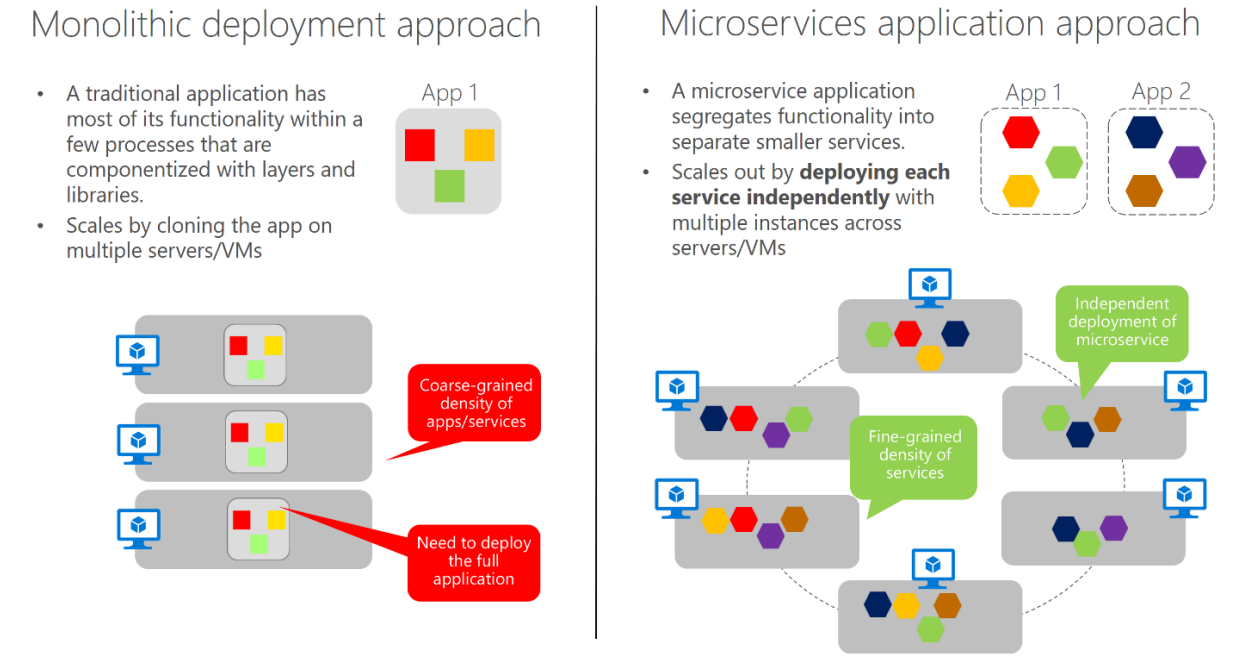
Microservices Architecture -

As the name implies, a microservices architecture is an approach to building a server application as a set of small services. That means a microservices architecture is mainly oriented to the back-end, although the approach is also being used for the front end. Each service runs in its own process and communicates with other processes using protocols such as HTTP/HTTPS, Websockets or AMQP. Each microservice implements a specific end-to-end domain or business capability within a certain context boundary and each must be developed autonomously and be deployable independently. Finally, each microservice should own its related domain data model and domain logic (sovereignty and decentralized data management) and could be based on different data storage technologies (SQL, NoSQL) and different programming languages.

What size should a microservice be? When developing a microservice, size shouldn't be the important point. Instead, the important point should be to create loosely coupled services so you have autonomy of development, deployment and scale for each service. Of course, when identifying and designing microservices, you should try to make them as small as possible as long as you don't have too many direct dependencies with other microservices. More important than the size of the microservice is the internal cohesion it must have and its independence from other services.

Why a microservices architecture? In short, it provides long-term agility. Microservices enable better maintainability in complex, large and high-scalable systems by letting you create applications based on many independently deployable services that each have granular and autonomous lifecycles.

As an additional benefit, microservices can scale out independently. Instead of having a single monolithic application that you must scale out as a unit., you can instead scale out specific microservices. That way you can scale just the functional area that needs more processing power or network bandwidth to support demand, rather than scaling out other areas of the application that don't need to be scaled. That means cost savings because you need less hardware.



As the figure shows, in the traditional monolithic approach, the application scales by cloning the whole app in several servers/VM. The microservices approach allows agile changes and rapid iteration of each microservice, because you can change specific, small areas of complex, large and scalable applications.
Architecting fine-grained microservices-based applications enables continuous integration and continuous delivery practices. It also accelerates delivery of new functions into the application. Fine grained composition of applications also allows you to run and test microservices in isolation, and to evolve them autonomously while maintaining clear contracts between them. As long as you don't change the interfaces or contracts, you can change the internal implementations of any microservice or add new functionality without breaking other microservices.
The following are important aspects to enable success in going into production with a microservices based system:

1. Monitoring and health checks of the services and infrastructure.
2. Scalable infrastructure for the services (that is, cloud and orchestrators).
3. Security design and implementation at multiple levels: authentication, authorization, secrets management, secure communication etc.
4. Rapid application delivery, usually with different teams focusing on different microservices.
5. DevOps and CI/CD practices and infrastructure.

Angular -

Client-Side framework like Angular -

For applications that use complex logic, developers had to put in extra effort to maintain separation of concerns for the app. Also jQuery did not provide facilities for data handling across views. Client-side frameworks allow one to develop advanced web applications like Single-Page-Application.

Angular Application Work -

Every Angular app consists of a file named angular.json. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find every point of the application. As shown below is the angular.json file format.

```
"Build":{
        "builder":"@angular-devkit/build-angular:browser",
        "Options":{
                "outputpath":"dist/angular-starter",
                "index":"src/index.html",
                "main":"src/main.ts",
                "polyfills":"src/polyfills.ts",
                "tsConfig":"tsconfig.app.json",
                "Aot":false,
                "Assets":[
                        "src/favicon.ico",
                        "src/assets"
                ],
                "Styles":[
```

```
                    "/node_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",
                    "src/style.css"
                ],
            }
}
```

Inside the build section, the main property of the options object defines the entry point of the application which in this case is main.ts.

The main.ts file creates a browser environment for the application to run and along with this, it also calls a function called bootstrapModule, which bootstraps the application, these two steps are performed in the following order inside the main.ts file.

```
import{platformBrowserDynamic} from '@angular/platform-browser-dynamic'
platformBrowserDynamic().bootstrapModule(AppModule)
```

In the above line of code AppModule is getting bootstrapped.

The appModule is declared in the app.module.ts file. This module contains declarations of all the components.

Example- app.module.ts file

```
Import {BrowserModule} from '@angular/platform-browser';
Import {ngModel} from '@angular/core';
Import {AppComponent} from './app.component';
@NgModule({
        Declarations:[
        AppComponent
        ],
        Imports:[
                BrowserModule
        ],
        Providers:[]
        entryComponents:[]
        bootstrap:[AppComponent]
})
Export class AppModule{}
```

As one can see in the above file, AppComponent is getting bootstrapped.

This component is defined in app.component.ts file. The file interacts with the webpage and serves data to it.

Example - app.component.ts

```
Import {Component} from '@angular/core';
@Component({
        Selector:'app-root',
        templateUrl:'./app.component.html',
        styleUrls:['./app.component.css']
})
Export class AppComponent{
        title='angular'
}
```

Each component is declared with three properties.

1. Selector - used for accessing the component
2. Template/TemplateURL - contains HTML of the component

3. StylesURL - Contains component-specific stylesheets

After this, Angular calls the index.html file. This file consequently calls the root component that is app-root. The root component is defined in app.component.ts

Example- index.html file

```
<!doctype html>
<html lang=”en”>
<head>
        <meta charset=”utf-8”>
        <title>Angular</title>
        <base href=”/”>
        <meta name=”viewport” content=”width=device-width,initial-scale=1”>
</head>
<body>
        <app-root></app-root>
</body>
</html>
```

The HTML template of the root component is displayed inside the <app-root> tags.

Advantages of Angular over other frameworks -

- Features that are provided out of the box- Angular provides a number of builtin features like routing, state management, rxjs library and http services straight out of the box. This means that one does not need to look for the above stated features separately. They are all provided with angular.
- Declarative UI - Angular uses HTML to render the UI of an application. HTML is a declarative language and is much easier to use than JavaScript.
- Long-term Google Support - Google announced Long-term support for Angular. This means that Google plans to stick with Angular and further scale up its ecosystem.

AngularJS	**Angular**
Architecture- AngularJS uses MVC or Model-View-Controller architecture, where the Model contains the business logic, Controller processes information and View shows the information present in the Model.	Architecture - Angular replaces controllers with Components. Components are nothing but directives with a predefined template.
Language - AngularJS uses javaScript language, which is dynamically typed language.	Language- Angular uses TypeScript language, which is statically typed language and is a superset of JavaScript. By using statically typed language, Angular provides better performance while developing larger applications.
Mobile support - AngularJS does not provide mobile support	Mobile Support - Angular is supported by all popular mobile browsers.
Structure - While developing larger applications, the process of maintaining code becomes tedious in the case of AngularJS.	In the case of Angular, it is easier to maintain code for larger applications as it provides a better structure.

Expression Syntax - While developing an AngularJS application, a developer needs to remember the correct ng-directive for binding an event or a property.	In Angular, property binding is done using "[]" attribute and event binding is done using "()" attribute.
Based on controllers concept	This is a component based UI approach.
Difficulty in SEO friendly application development	Ease to create SEO friendly applications.

AOT Compilation -

Every Angular application consists of components and templates which the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser.

Angular provides two types of compilation -

1. JIT (just-in-Time) compilation - The application compiles inside the browser during runtime.
   Ng build
   Ng serve
2. AOT (Ahead-of-time) compilation - the application compiles during the build time.
   Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to faster rendering. The compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to fewer ajax requests. Developers can detect and handle errors during the building phase, which helps in minimizing errors.
   The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide better security to the application.
   Ng build –aot
   Ng serve –aot

Key components of Angular-

1. Component - these are the basic building blocks of angular application to control HTML views.
2. Modules - An angular module is set of angular basic building blocks like component, directives, services etc. An application is divided into logical pieces and each piece of code is called as "module" which perform a single task.
3. Templates - This represent the views of an angular application.
4. Services - it is used to create components which can be shared across the entire application.
5. Metadata -This can be used to add more data to an Angular class.

Component	**Directive**
To register a component we use	To register directives we use @Directive

@Component meta-data annotation.	meta-data annotation.
Components are typically used to create UI widgets	Directives is used to add behaviour to an existing DOM element
Component is used to break up the application into smaller components	Directive is used to design reusable components.
Only one component can be present in DOM element.	Many directives can be used per DOM element.
@View decorator or templateurl/template are mandatory.	Directives doesn't use View.

Components, Modules and Services in Angular -

Components -

In Angular, Components are the basic building blocks, which control part of the UI for any application.

A component is defined using the @Component decorator. Every component consists of three parts, the template which loads the view for the component, a stylesheet which defines the look and feel for the component and a class that contains the business logic for the component.

```
Ng g c test
```

One can see the generated component inside the src/app/test folder. The component will be defined inside test.component.ts file.

```
Import {Component, OnInit} from '@angular/core';
@Component({
        Selector:'app-test',
        templateUrl:'./test.component.html',
        styleUrls:['./test.component.css']
})
Export class TestComponent implements OnInit{
        constructor() {}
        ngOnInit() {}
}
```

Modules -

A module is a place where we can group components, directives, services and pipes. Module decides whether the components, directives etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @ngModule decorator.

By default modules are two types -

1. Root Module
2. Feature Module

Every application can have only one root module whereas, it can have one or more feature modules.

A root module imports browserModule, whereas a feature module imports CommonModule.

The root module is defined inside app.module.ts -

```
Import {BrowserModule} from '@angular/platform-browser';
```

```
Import {NgModule} from '@angular/core';
Import {AppComponent} from './app.component';
Import {testComponent} from './test.test.component';
@NgModule({
        Declarations: [
                AppComponent,
                TestComponent
        ],
        Imports: [
                BrowserModule
        ],
        Providers:[],
        bootstrap:[AppComponenent]
})
Export class AppModule{}
```

To create feature module, run the following command -

```
Ng g m test-module
```

The module is created inside the src/app/test-module/test-module.module.ts file-

```
Import {NgModule} from '@angular/core';
Import {CommonModule} from '@angular/common';
@NgModule({
        Declarations:[],
        Imports:[
                CommonModule
        ]
})
Export class testModuleModule{}
```

Services-

Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an angular application.

A service is defined using a @Injectable decorator. A function defined inside a service can be invoked from any component or directive.

To create a service, run the following command.

```
Ng g s test-service
```

The service will be created inside src/app/test-service.service.ts file-

```
Import {Injectable} from @angular/core';
@injectable({
        providedIn:'root'
})
Export class testServiceService{
        constructor() {}
}
```

Any method/function defined inside the TestServiceService class can be directly used inside any component by just importing the service.

Lifecycle hooks in Angular -

Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides hooks to tap into these phases and trigger changes at specific phases in the lifecycle.

1. ngOnChanges()-
   This hook/method is called before ngOnInit and whenever one or more input properties of the component changes.
   This method/hook receives a SimpleChanges object which contains the previous and current values of the property.
2. ngOnInit()-
   This hook gets called once, after the ngOnChanges hook. It initializes the component and sets the input properties of the component.
3. ngDoCheck()-
   It gets called after ngOnChanges and ngOnInit and is used to detect and act on changes that cannot be detected by Angular. We can implement our change detection algorithm in this hook. ngAfterContentInit() it gets called after the first ngDoCheck hook. This hook responds after the content gets projected inside the component.
4. ngAfterContentinit() -
   This is called in response after Angular projects external content into the component's view.
5. ngAfterContentChecked()-
   It gets called after ngAfterContentInit and every subsequent ngDoCheck. It responds after the projected content is checked.
6. ngAfterViewInit()-
   It responds after a components view or a child component's view is initialized.
7. ngAfterViewChecked()-
   It gets called after ngAfterViewInit and responds after the component's view or the child component's view is checked.
8. ngOnDestroy()-
   It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach handlers.

String interpolation and property binding in Angular -

String interpolation and property binding are parts of data-binding in angular.
Data-binding is a feature in angular, which provides a way to communicate between the component (model) and its view (HTML template).
Data-binding can be done in two ways, one-way binding and two-way binding.
In Angular, data from the component can be inserted inside the HTML template. In one-way binding, any changes in the component will directly reflect inside the HTML template but vice-versa is not possible. Whereas, it is possible in two-way binding.
String interpolation and property binding allow only one-way data binding.
String interpolation uses the double curly braces {{ }} to display data from the component Angular automatically runs the expression written inside the curly braces, for example, {{ 2+2 }} will be evaluated byAngular and the output 4, will be displayed inside the HTML template. Using

property binding, we can bind the DOM properties of an HTML element to a components property. Property binding uses the square brackets [ ] syntax.

Promise	**Observable**
Promise is eager.	Observable is lazy.
Emits a single value	Emits multiple values over a period of time.
Not Lazy	Lazy. An observable is not called until we subscribe to the observable.
Cannot be canceled	Can be canceled by using the unsubscribe() method
Promises are always asynchronous. Even when the promise is immediately resolved.	Observable can be both synchronous and asynchronous.
	Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc.

Consider the following Observable-

```
Const observable = rxjs.Observable.create(observer=>{
        console.log('The inside an observable');
        observer.next('Hello World');
        observer.complete();
})
console.log('before subscribing an observable');
observable.subscribe((message)=>console.log(message));
```

output->

```
before subscribing an observable
The inside an observable
Hello World
```

As you can see, observables are lazy. Observables runs only when someone subscribes to them.

Consider a Promise -

```
Const promise = new Promise((resolve,reject)=>{
        console.log('text inside promise');
        resolve('Hello World');
});
console.log('Before calling promise');
greetingPoster.then(message=>console.log(message));
```

output->

```
text inside promise
Before calling promise
Hello World
```

As you can see the message inside Promise is displayed first. This means that a promise runs before the then method is called. Therefore , promises are eager.

Directives in Angular -

A directive is a class in Angular that is declared with a @Directive decorator.

Every directive has its own behavior and can be imported into various components of an application.
Consider an application, where multiple components need to have similar functionalities. The norm thing to do is by adding this functionality individually to every component but, this task is tedious to perform. In such a situation, one can create a directive having the required functionality and then, import the directive to components which require this functionality.
Types of directives-

1. Component directives -
   These form the main class in directives. Instead of @Directive decorator we use @Component decorator to declare these directives. These directives have a view, a stylesheet and a selector property.
2. Structural directives -
   These directives are generally used to manipulate DOM elements. Every structural directive has a '*' sign before them. We can apply these directives to any DOM element.

```
<div *ngIf="isready" class="display_name"> {{name}} </div>
<div class="details" *ngFor='let x in details'>
        <p>{{x.name}}</p>
        <p>{{x.address}}</p>
        <p>{{x.age}}</p>
</div>
```

   In the above example, we can *ngIf and *ngFor directives being used.
   *ngIf is used to check a boolean value and if it's truthy, the div element will be displayed.
   *ngFor is used to iterate over a list and display each item of the list.
3. Attribute directives -
   These directives are used to change the look and behaviour of a DOM element.
   To create attribute directive use below command

```
Ng g directive blueBackground
Import {Directive, ElementRef} from '@angular/core'
@Directive({
        selector:'[appBlueBackground]'
})
Export class BluebackgroundDirective{
        constructor(el:ElementRef){
                el.nativeElement.style.backgroundColor="blue";
        }
}
```

   Now we can apply the above directive

```
<p appBlueBackground>Hello World</p>
```

How does one share data between components in Angular?

Parent Component → Child Component

Parent to Child using @input decorator.

Consider the following parent component

```
@Component({
        Selector:'app-parent',
        template:'<app-child[data]=data></app-data>',
```

```
        styleUrls:[''./parent.component.css]
})
Export class parentComponent{
        data.string="Message from parent";
        constructor() {}
}
```

In the above parent component, we are passing "data" property to the following child component.

```
Import {Component, Input} from '@angular/core'
@Component({
        Selector:'app-child',
        template:'<p>{{data}}</p>',
        styleUrls:['./child.component.css']
})
Export class ChildComponent{
        @Input() data:string
        constructor() {}
}
```

In the child component, we are using @Input decorator to capture data coming from a parent component and using it inside the child component's template.

Parent Component ← Child Component

Child to parent using @ViewChild decorator.

Child component-

```
Import {Component} from '@angular/core';
@Component({
        Selector:'app-child',
        template:'<p>{{data}}</p>',
        styleurls:['./child.component.css']
})
Export class ChildComponent{
        data.string="Message from child to parent";
        constructor(){}
}
```

Parent Component-

```
Import {Component, ViewChild, AfterViewInit} from '@angular/core';
Import {ChildComponent} from './../child/child.component';
@Component({
        Selector:'app-parent',
        template:'<p>{{dataFromChild}}</p>',
        styleurls:['./parent.component.css']
})
Export class parentComponent implements AfterViewInit{
        datafromChild: string;
        @ViewChild(ChildComponent,{static:false}) child;
        ngAfterViewInit(){
                this.dataFromChild=this.child.data;
        }
        constructor(){}
```

```
}
```

In the above example, a property named “data” is passed from the child component to the parent component.

@ViewChild decorator is used to reference the child component as “child” property.

Using the ngAfterViewInit hook, we assign the child’s data property to the messageFromChild property and use it in the parent component’s template.

Child to parent @output and EventEmitter

In this method, we bind a DOM element inside the child component, to an event and using this event we emit data that will captured by the parent component:

Child Component-

```
Import {Component, Output, EventEmitter} from ‘@angular/core’;
@Component({
        Selector:’app-child’,
        template:’<button (click)=”emitData()”>Click to emit data</button>’,
        styleUrls:[‘./child.component.css’]
})
Export class ChildComponent{
        data:string=”Message from child to parent”;
        @output() dataEvent =new EventEmitter<string>();
        constructor(){}
        emitData(){
                this.dataEvent.emit(this.data);
        }
}
```

As you can see in the child component, we have used @Output property to bind an EventEmitter. This event emitter emits data when the button in the template is clicked.

In the parent component’s template can be capture the emitted data like this:

```
<app-child (dataEvent)=”receiveData($event)”><app-child>
```

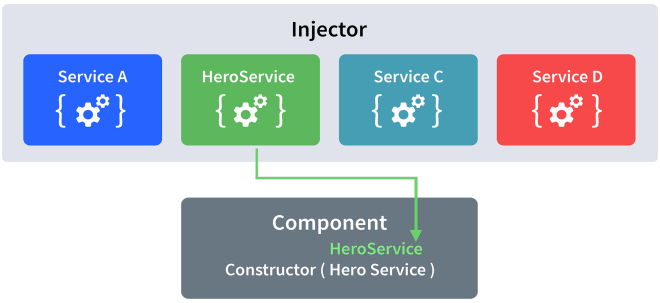
Then inside the receiveData function we can handle the emitted data:

```
receiveData($event){
        this.dataFromChild=$event;
}
```

Dependency injection in Angular -

Dependency injection is an application design pattern which is implemented by Angular.

Dependencies in angular are nothing but services which have a functionality. Functionality of a service can be needed by various components and derivatives in an application. Angular provides a smooth mechanism by which we can inject these dependencies in our components and directives. So basically, we are just making dependencies which are injectable across all components of an application.



Consider the following service, which can be generated using -

```
Ng g service test
Import {Injectable} from '@angular/core';
@Injectable({
        providedIn:'root'
})
Export class TestService{
        importantValue:number=42;
        constructor() {}
        returnimportantValue(){
                Return this.importantValue;
        }
}
```

As one can notice, we can create injectable dependencies by adding the @Injectable decorator to a class.

We inject the above dependency inside the following component:

```
Import {TestService} from './../test.service';
Import {Component, OnInit} from '@angular/core';
@Component({
        Selector:'app-test',
        templateUrl:'./test.component.html',
        styleUrls:['./test.component.css']
})
Export class TestComponent implements OnInit{
        Value:number;
        constructor(private testservice:TestService){}
        ngOnInit(){
                this.value=this.testService.returnImportantValue();
        }
}
```

One can see we have imported our TestService at the top of the page. Then, we have created an instance inside the constructor of the component and implemented the returnImportantValue function of the service. From the above example, we can observe how angular provides a smooth way to inject dependencies in any component.

MVVM architecture -

1. Model -
   Model contains the structure of an entity. In simple terms it contains data of an object.
2. View -
   View is the visual layer of the application. It displays the data contained inside the Model. In angular terms, this will be the HTML template of an component.
3. ViewModel -
   ViewModel is an abstract layer of the application. A viewmodel handles the logic of the application. It manages the data of a model and displays it in the view.
   View and ViewModel are connected with data-binding (two-way data-binding in this case). Any change in the view, the viewmodel takes a note and changes the appropriate data inside the model.

Difference between constructor and ngOnInit -

TypeScript classes have a default method called constructor which is normally used for the initialization purpose. Whereas ngOnInit method is specific to Angular, especially used to define Angular bindings.Even though constructor getting called first, it is preferred to move all of your angular bindings to ngOnInit method, In order to use ngOnInit, you need to implement interface as below.

```
Export class App implements OnInit{
        constructor(){
                //called first time before the ngOnInit
        }
        ngOnInit(){
                //Called after the constructor and called after the first ngOnChanges()
        }
}
```

Closure in JavaScript

Hoisting in javascript
Javascript array add and delete element
Scopes in javascript
Typescript /javascript component level iterative methods
Arrow function in javascript/ typescript
Difference between map and filter in angular
Difference between concat and join in angular
Route guard in angular
Template reference variables in angular
Host listener and host binding in angular
Pure pipes and impure pipes in angular

Positions in css
Pseudo classes in angular
Blocked element and inline element in angular

Common table expressions in sql
MVC models and Angular components difference
Regular expressions in C#
Repository design pattern
MVC application performance improve
Serialization in C#
Concurrency how to handle
C# extension methods
String vs StringBuilder
Mutable vs immutable
ViewResult in MVC
Routing in MVC
Routing in Angular
Angular Lifecycle hooks
Angular Route guard
Jquery and javascript
Delegate
Generics
How to pass one page to another page data
Write 1 Linq query
Repository design pattern
Singleton design pattern
Solid principles with example
Entity framework
Tell me complex code part which u done
C# virtual override new keyword
MVC filters how to apply and why
MVC rest API how to call and its structure.
Angular and MVC output files after build
In C# using keyword
.net core concepts
Extension Methods in C#
What is serialization and deserialization
Async and await why uses
Two way data binding in angular

What is typescript?
And- TypeScript is strongly typed language meaning that everything has data types because of strongly typing, typescript has great tooling including inline documentation, syntax checking, code navigation and advanced refactoring, so typescript helps us better reason about our code.

And typescript ES6 class based object orientation plus more. It implements classes, interfaces and inheritance.

What in Package.json file in Angular?
Ans - Package.json file that lists each package we need for our angular application. We can also specify the desired version of each package. We can also specify the desired version of each package. We then tell npm to use the package.json file to install all of the defined packages along with their dependencies. In the package.json file the list of packages is divided into two parts. (1) The dependencies list is for the packages we need for development that must also be deployed. The dependencies list includes the primary Angular packages along with supporting packages, such as RxJS, for working with data. (2) The devDependencies list is for the packages we only need for development. The devDependencies include the Angular CLI. TypeScript is here in the devDependencies since we transpiled the code to javaScript before deployment and many of these are for unit code tests. By defining a package.json file for our Angular application, we ensure everyone on the team the appropriate packages and versions.

What is component in Angular?
Ans - An Angular component includes a template. Which lays out the user interface fragment defining a view for the application. It is created with HTML and defines what is rendered on the page. We use Angular bindings and directives in the HTML to power up the view. Components is the combined with class and metadata decorator.