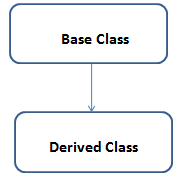
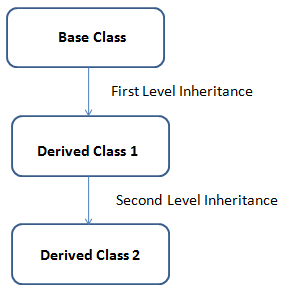
1. **OOPS Basics**:- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. For example, a person is an object which has certain properties such as height, gender, age, etc.
2. **Inheritance −** The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods. For example, if we have a class Vehicle that has properties like Color, Price etc, we can create 2 classes like Bike and Car from it that have those 2 properties and additional properties that are specialized for them like a car has number Of Windows while a bike cannot. Same is applicable to methods.

Types Of Inheritance.

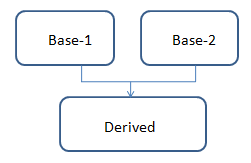
1. Single inheritance.



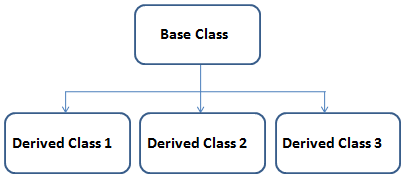
1. Multi-level inheritance.



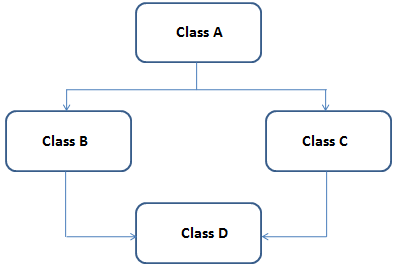
3.Multiple inheritance.



4.Hierarchical Inheritance.



5.Hybrid Inheritance.

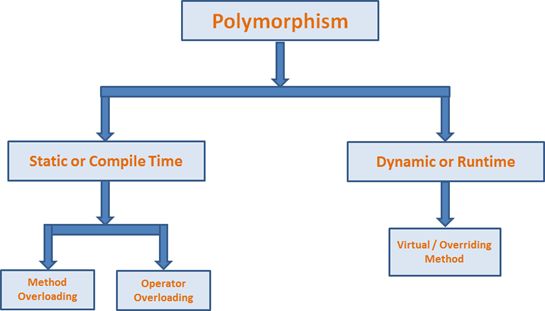


**Polymorphism** :- Polymorphism is a Greek word, meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to a class to have multiple implementations with the same name.

Types of Polymorphism

There are two types of polymorphism in C#:

* Static / Compile Time Polymorphism.
* Dynamic / Runtime Polymorphism.



**Constructure** :- A special method of the class that is automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize the private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor of the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

### Some of the key points regarding constructor are

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor can not be a parametrized constructor.
* Within a class, you can create one static constructor only.

### In C#, constructors can be divided into 5 types

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

**Static Keyword:-static**is a modifier in C# which is applicable for the following:

1.Class

2.Variable

3.Mathods

4.Consturctor

It is also applicable to property, event, and operators. To create a static member(class, variable, methods, constructor), precede its declaration with the keyword *static*. When a member is declared static, it can be accessed with the name of its class directly.

#### 1.Static Class

A static class is declared with the help of *static*keyword. A static class can only contain static data members, static methods, and a static constructor. It is not allowed to create objects of the static class. Static classes are sealed, means one cannot inherit a static class from another class.

#### 2.Static Variable

A static variable is declared with the help of static keyword. When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are accessed with the name of the class, they do not require any object for access.

#### 3.Static Method

A static method is declared with the help of static keyword. Static methods are accessed with the name of the class.

#### 4.Static Constructor

A static constructor is declared with the help of static keyword. Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class. A static constructor is initialized static fields or data of the class and to be executed only once.

**Points To Remember:**

1. It can’t be called directly.

2. When it is executing, then the user has no control.

3. It does not take access modifiers or any parameters.

4. It is called automatically to initialize the class before the first instance created.

**Diff Between Static and Normal Class**

The following is the difference between a static and non-static class −

1. Non-static classes can be instantiated, whereas static classes cannot be instantiated i.e. you cannot use the new keyword to create a variable of the class type
2. Static classes can only have static methods.
3. Non-static classes can have instance method and static methods.
4. access the members of a static class by using the class name itself
5. Static class is sealed.

Static class can not be inherited but Normal class can be inherited.

**Static method & Non-static method**

A **static method** belongs to the class and a **non-static method** belongs to an object of a class. The static methods can by accessed directly from the class, while non-static methods (or instance methods as I like to call them) have to be accessed from an instance.

**Sealed class**:-Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the **sealed** keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

The following is the **syntax** of a sealed class :

sealed class class\_name

{

// data members

// methods

.

}

Example:-

using System; // Sealed class

sealed class SealedClass {

    // Calling Function

    public int Add(int a, int b)

    {

        return a + b;

    }

}

class Program {

    // Main Method

    static void Main(string[] args)

    {

        // Creating an object of Sealed Class

        SealedClass slc = new SealedClass();

        // Performing Addition operation

        int total = slc.Add(6, 4);

        Console.WriteLine("Total = " + total.ToString());

    }

}

**Abstraction:**  An abstract class is an incomplete class or special class we can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.

## C# Abstract Class Features

1. An abstract class can inherit from a class and one or more interfaces.
2. An abstract class can implement code with non-Abstract methods.
3. An Abstract class can have modifiers for methods, properties etc.
4. An Abstract class can have constants and fields.
5. An abstract class can implement a property.
6. An abstract class can have constructors or destructors.
7. An abstract class cannot be inherited from by structures.
8. An abstract class cannot support multiple inheritance.

using System;

namespace MyApplication

{

// Abstract class

abstract class Animal

{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep()

{

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

}

**Interface**:-Interface in C# is **a blueprint of a class**. It is like abstract class because all the methods which are declared inside the interface are abstract methods. ... It is used to achieve multiple inheritance which can't be achieved by class. It is used to achieve fully abstraction because it cannot have method body.

using System;

namespace MyApplication

{

// Interface

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

}

Constructors:- A special method of the class that is automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize the private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor of the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

### Some of the key points regarding constructor are

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor can not be a parametrized constructor.
* Within a class, you can create one static constructor only.

### In C#, constructors can be divided into 5 types

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

## Default Constructor in C#

A constructor without any parameters is called a default constructor; in other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class with different values. The default constructor initializes:

using System;

namespace DefaultConstractor

{

class addition

{

int a, b;

public addition() //default contructor

{

a = 100;

b = 175;

}

public static void Main()

{

addition obj = new addition(); //an object is created , constructor is called

Console.WriteLine(obj.a);

Console.WriteLine(obj.b);

Console.Read();

}

}

}

## Parameterized Constructor in C#

A constructor with at least one parameter is called a parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

using System;

namespace Constructor

{

class paraconstrctor

{

public int a, b;

public paraconstrctor(int x, int y) // decalaring Paremetrized Constructor with ing x,y parameter

{

a = x;

b = y;

}

}

class MainClass

{

static void Main()

{

paraconstrctor v = new paraconstrctor(100, 175); // Creating object of Parameterized Constructor and ing values

Console.WriteLine("-----------parameterized constructor example by vithal wadje---------------");

Console.WriteLine("\t");

Console.WriteLine("value of a=" + v.a );

Console.WriteLine("value of b=" + v.b);

Console.Read();

}

}

}

# C# - Data Types

C# is a strongly-typed language. It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.

using System;

public class Program

{

public static void Main()

{

string stringVar = "Hello World!!";

int intVar = 100;

float floatVar = 10.2f;

char charVar = 'A';

bool boolVar = true;

Console.WriteLine(stringVar);

Console.WriteLine(intVar);

Console.WriteLine(floatVar);

Console.WriteLine(charVar);

Console.WriteLine(boolVar);

}

}

C# DataTypes



## Predefined Data Types in C#

C# includes some predefined value types and reference types. The following table lists predefined data types:

| Type | Description | Range | Suffix |
| --- | --- | --- | --- |
| Byte | 8-bit unsigned integer | 0 to 255 |  |
| Sbyte | 8-bit signed integer | -128 to 127 |  |
| Short | 16-bit signed integer | -32,768 to 32,767 |  |
| Ushort | 16-bit unsigned integer | 0 to 65,535 |  |
| Int | 32-bit signed integer | -2,147,483,648 to 2,147,483,647 |  |
| Uint | 32-bit unsigned integer | 0 to 4,294,967,295 | u |
| Long | 64-bit signed integer | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | l |
| Ulong | 64-bit unsigned integer | 0 to 18,446,744,073,709,551,615 | ul |
| Float | 32-bit Single-precision floating point type | -3.402823e38 to 3.402823e38 | f |
| double | 64-bit double-precision floating point type | -1.79769313486232e308 to 1.79769313486232e308 | d |
| decimal | 128-bit decimal type for financial and monetary calculations | (+ or -)1.0 x 10e-28 to 7.9 x 10e28 | m |
| Char | 16-bit single Unicode character | Any valid character, e.g. a,\*, \x0058 (hex), or\u0058 (Unicode) |  |
| Bool | 8-bit logical true/false value | True or False |  |
| Object | Base type of all other types. |  |  |
| String | A sequence of Unicode characters |  |  |
| DateTime | Represents date and time | 0:00:00am 1/1/01 to 11:59:59pm 12/31/9999 |  |

The key difference between varchar and nvarchar is the way they are stored, varchar is stored as regular 8-bit data(1 byte per character) and nvarchar stores data at 2 bytes per character. Due to this reason, nvarchar can hold upto 4000 characters and it takes double the space as SQL varchar.

Difference between VARCHAR and NVARCHAR are following:

1.       **Data Store :**A VARCHAR column is restricted to an 8-bit codepage while An NVARCHAR column can store any Unicode data.

2.       **Performance :** A VARCHAR column is slow to read or write to the database while A NVARCHAR is fast because all modern operating systems and development platforms use Unicode internally. By using NVARCHAR rather than VARCHAR, you can avoid doing encoding conversions every time you read from or write to the database. Conversions take time, and are prone to errors. And recovery from conversion errors is a non-trivial problem.

3.       **Size :** VARCHAR can hold up to 2^31 bytes of data, or 2,147,483,648 characters; NVARCHAR(MAX) can hold 2^30 bytes, or 1,073,741,823 characters.

4.       **Memory :** NVARCHAR take twice space compare VARCHAR.

[**Int32**](https://www.geeksforgeeks.org/c-sharp-int32-struct/?ref=rp)**:**This Struct is used to represents 32-bit signed integer. The **Int32**can store both types of values including negative and positive between the ranges of **-2147483648 to +2147483647**.

[**Int64**](https://www.geeksforgeeks.org/c-sharp-int-64-struct/?ref=rp)**:**This Struct is used to represents 64-bit signed integer. The **Int64**can store both types of values including negative and positive between the ranges of **-9,223,372,036,854,775,808 to +9, 223,372,036,854,775,807.**

## Decimal vs Double vs Float

The Decimal, Double, and Float variable types are different in the way that they store the values. Precision is the main difference where float is a single precision (32 bit) floating point data type, double is a double precision (64 bit) floating point data type and decimal is a 128-bit floating point data type.

## Double

Double Types are probably the most normally used data type for real values, except handling money.

## Float

It is used mostly in graphic libraries because very high demands for processing powers, also used situations that can endure rounding errors.

# ASP.NET MVC Architecture

The MVC architectural pattern has existed for a long time in software engineering. All most all the languages use MVC with slight variation, but conceptually it remains the same.

Let's understand the MVC architecture supported in ASP.NET.

MVC stands for Model, View, and Controller. MVC separates an application into three components - Model, View, and Controller.

**Model**: Model represents the shape of the data. A class in C# is used to describe a model. Model objects store data retrieved from the database.

**Model represents the data**.

**View**: View in MVC is a user interface. View display model data to the user and also enables them to modify them. View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor syntax) that makes it easy to communicate with the model and the controller.

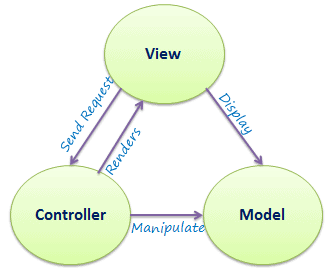
**View is the User Interface.**

**Controller**: The controller handles the user request. Typically, the user uses the view and raises an HTTP request, which will be handled by the controller. The controller processes the request and returns the appropriate view as a response.

Selecting MVC Project as Empty creates basic folder structure for the project. You will also see two View Engine ASPX and Razor. Razor view engine is new and improved view engine introduced in ASP.NET MVC 3. In Solution Explorer, you will see the following three main architecture components of ASP.NET MVC project that is Model, View and Controller.

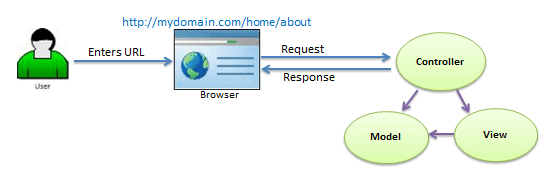
**Controller is the request handler.**

The following figure illustrates the interaction between Model, View, and Controller.

[](https://www.tutorialsteacher.com/Content/images/mvc/mvc-architecture.png)

MVC Architecture

The following figure illustrates the flow of the user's request in ASP.NET MVC.

[](https://www.tutorialsteacher.com/Content/images/mvc/request-handling-in-mvc.png)

Request Flow in MVC Architecture

As per the above figure, when a user enters a URL in the browser, it goes to the webserver and routed to a controller. A controller executes related view and models for that request and create the response and sends it back to the browser.

 Points to Remember

1. MVC stands for Model, View and Controller.
2. Model represents the data
3. View is the User Interface.
4. Controller is the request handler.

Asp.net Core:

It’s a web Framework in C#;ASP.NET Core is the new web framework from Microsoft. ASP.NET Core is the framework you want to use for web development with .NET.

**Razor Syntax** is a powerful and easiest way to write server-based code directly into your view pages. Razor is a markup syntax that allows you to embed C# Programming directly into your view page. It is generally written in the **.cshtml** file.

**1.** Use **@{ … }** block to write C# code.

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. for(int i=0; i<5; i++)
11. {
12. <h3 style="color:deeppink">**Hello** world @i</h3>
13. }
14. }
15. </div>

**2.** All the Inline Expressions like Variables and Functions **starts with @**.

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. int i = 10;
11. <h2>@i</h2>
13. <h3><i>**Today** **Date** is @DateTime.**Now**.**ToShortDateString**()</i></h3>
15. }
16. </div>

**3.** Variables are declared with the **var** keyword and all the code statements end with a **semicolon(;)**.

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var a = 5;
11. var b = 10;
12. var result = a + b;
13. <h2>**Addition** of @a + @b = @result</h2>
15. }
16. </div>

**4.** Keep String value within **double quotes (" ")**

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var str = "Hello World";
11. <h3>@str</h3>
12. }
13. </div>

**5.** Use HTML Markup for adding HTML code inside Razor block.

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var str = "Hello World";
11. <h3>@str</h3>
12. <span>**This** is **Text** inside **Razor**.</span>
13. }
14. </div>

**6.** Use **Double @@**, if you want to print **single @**.

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var price = 30;
11. <h2>**You** can get 1GB **Data** @@ of @price</h2>
12. }
13. </div>

**7.** Control Statement: if, else if and else

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var price = 45;
11. if(@price > 30)
12. {
13. <h3>**Price** is **Higher** than 30.</h3>
14. }
15. else if(@price == 30)
16. {
17. <h3>**Price** is **Equal** to 30.</h3>
18. }
19. else
20. {
21. <h3>**Price** is **Lower** than 30.</h3>
22. }
23. }
24. </div>

**8.** Switch Case

**Example:**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. var **Days** = **DateTime**.**Now**.**DayOfWeek**.**ToString**();
12. switch (**Days**)
13. {
14. case "Sunday": <strong>**Its** **Holiday**</strong>
15. break;
16. case "Monday": <strong>**Oh** no! **It**'s Monday Again</strong>
17. break;
18. case "Tuesday": <strong>Go to Work. It's **Tuesday**</strong>
19. break;
20. case "Wednesday": <strong>**Continue** to **Work**. **It**'s Wednesday</strong>
21. break;
22. case "Thursday": <strong>Continue to Work. It's **Thursday**</strong>
23. break;
24. case "Friday": <strong>**Yeah**! **It**'s Friday.</strong>
25. break;
26. case "Saturday": <strong>Hurreyy... It's **Saturday**.</strong>
27. break;
28. default: <strong>**Its** **Mystrious**. I don't know the day name.</strong>
29. break;
30. }
31. }
33. </div>

**9.** Loop Statement: for, foreach, while, do while

**Example:**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. <h3>**For** **Loop** **Example**</h3>
11. for (var i = 0; i < 5; i++)
12. {
13. <strong>@i </strong>
14. }
16. <h3>**While** **Loop** **Example**</h3>
17. var j = 0;
18. while(j<5)
19. {
20. <strong>@j </strong>
21. j++;
22. }
24. <h3>**Do** **While** **Loop** **Example**</h3>
25. var k = 0;
26. do
27. {
28. <strong>@k </strong>
29. k++;
30. }
31. while (k < 5);
32. }
34. </div>

**10.** Foreach and Array

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. int[] arr = { 1, 3, 5, 7, 11, 13, 17, 19 };
11. foreach (int x in arr)
12. {
13. <span>@x, </span>
14. }
15. }
17. </div>

**11.** Handling Errors with Try Catch

**Example**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @{
10. try
11. {
12. int a, b, result;
13. a = 5;
14. b = 0;
15. result = a / b;
16. <span>@result </span>
17. }
18. catch(**Exception** ex)
19. {
20. <span>@ex.**ToString**()</span>
21. }
22. }
24. </div>

**12.** **@using** : Add NameSpace

You can add namespace by using the**@using** command.

**Example:**

1. @{
2. **ViewBag**.**Title** = "Computer Shop Management";
3. }
5. <div class="jumbotron">
6. <h2 style="color:chocolate">**Welcome** to **Computer** **Shop** **Management**</h2>
7. </div>
8. <div>
9. @using **System**.IO;
10. @using **System**.**Data**.**SqlClient**;
11. @using **System**.**Configuration**;
12. @{
13. // Your Code Here.
14. }
16. </div>

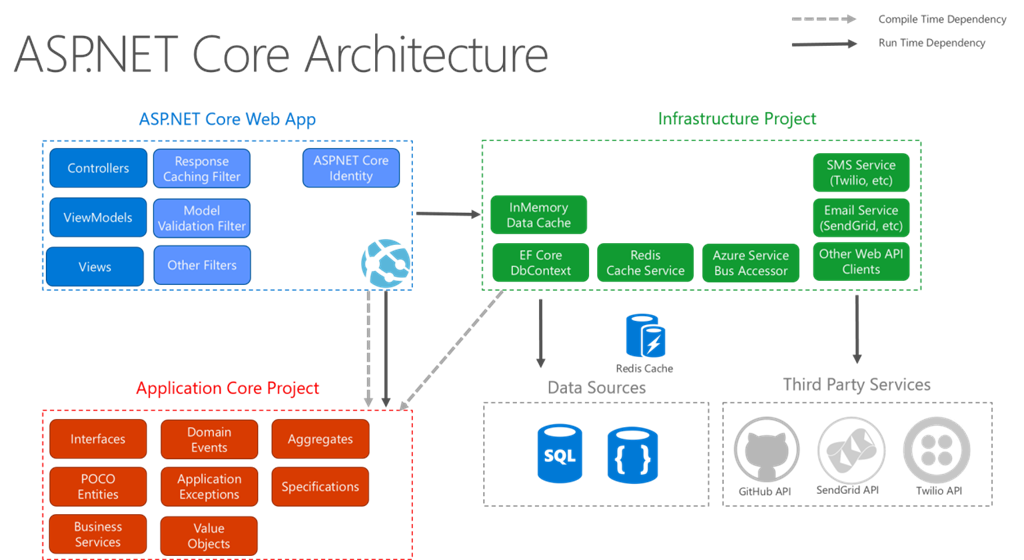
**13.**Comments in Razor

1. @\*
2. @{
3. /\* C# comment \*/
4. // Another C# comment
5. }
6. <!-- HTML comment -->

## ASP.NET Core

[ASP.NET](https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet) is a popular web-development framework for building web apps on the [.NET platform](https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet).

ASP.NET Core is the open-source version of ASP.NET, that runs on macOS, Linux, and Windows. ASP.NET Core was first released in 2016 and is a re-design of earlier Windows-only versions of ASP.NET.



Asp.net Core Benefit

ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, or Linux.

ASP.NET Core provides the following benefits:

* A unified story for building web UI and web APIs.
* Architected for testability.
* [Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-5.0) makes coding page-focused scenarios easier and more productive.
* [Blazor](https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0) lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
* Ability to develop and run on Windows, macOS, and Linux.
* Open-source and [community-focused](https://live.asp.net/).
* Integration of [modern, client-side frameworks](https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0) and development workflows.
* Support for hosting Remote Procedure Call (RPC) services using [gRPC](https://docs.microsoft.com/en-us/aspnet/core/grpc/?view=aspnetcore-5.0).
* A cloud-ready, environment-based [configuration system](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-5.0).
* Built-in [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0).

A JSON file is a file that **stores simple data structures and objects in JavaScript Object Notation** (JSON) format, which is a standard data interchange format. It is primarily used for transmitting data between a web application and a server.

*Startup.cs* file -This class is described by its name: startup. It is the entry point of the application. It configures the request pipeline which handles all requests made to the application. The inception of startup class is in OWIN (Open Web Interface for.NET) application that is specification to reduce dependency of application on server.

web.config file is a xml based configuration file used in ASP.NET based application to manage various settings that concerned with configuration of our website. In this way we can separate our application logic from configuration logic. And the main benefit of this is, if we want to change some configuration settings then we do not need to restart our application to apply new changes, ASP.NET automatically detects the changes and applies them to the running ASP.NET application.

The ASP.NET framework uses a hierarchical configuration system. You can place a web.config file in any subdirectory of an application. The file then applies to any pages located in the same directory or any subdirectories.

Diff Between .net and .net Core.

**.NET Framework**:

* Do not have time to learn new technology.
* Need a stable environment to work in.
* Have nearer release schedules.
* We are already working on an existing app and extending its functionality.
* Already have an existing team with .NET expertise and building production-ready software.
* Do not want to deal with continuous upgrades and changes.
* Building Windows client applications using Windows Forms or WPF

**.NET Core**:

* Want to target your apps on Windows, Linux, and Mac operating systems.
* Are not afraid of learning new things.
* Are not afraid of breaking and fixing things since .NET Core is not fully matured yet.
* A student who is just learning .NET.
* Love open source.

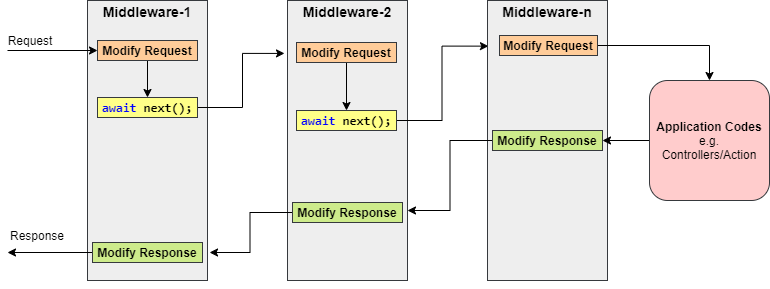
## What is Middleware?

Middleware is a piece of code in an application pipeline used to handle requests and responses.

For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc.

Middleware can be built-in as part of the .NET Core framework, added via NuGet packages, or can be custom middleware. These middleware components are configured as part of the application startup class in the configure method. Configure methods set up a request processing pipeline for an ASP.NET Core application. It consists of a sequence of request delegates called one after the other.

The following figure illustrates how a request process through middleware components.



Generally, each middleware may handle the incoming requests and passes execution to the next middleware for further processing.

But a middleware component can decide not to call the next piece of middleware in the pipeline. This is called short-circuiting or terminate the request pipeline. Short-circuiting is often desirable because it avoids unnecessary work. For example, if the request is for a static file like an image CSS file JavaScript file etc., these static files middleware can handle and serve that request and then short-circuit the rest of the pipeline.

Let’s create an ASP.NET Core Web application and observe the default configuration of middleware in the Configure method of the Startup class.

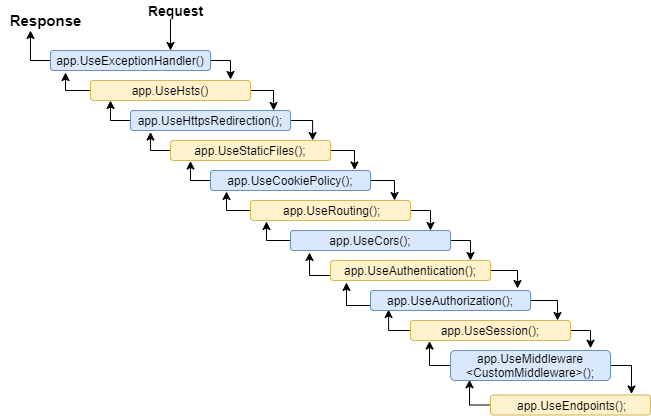
1. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
2. {
3. **if** (env.IsDevelopment())
4. {
5. //This middleware is used reports app runtime errors in development environment.
6. app.UseDeveloperExceptionPage();
7. }
8. **else**
9. {
10. //This middleware is catches exceptions thrown in production environment.
11. app.UseExceptionHandler("/Error");
12. // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
13. app.UseHsts(); //adds the Strict-Transport-Security header.
14. }
15. //This middleware is used to redirects HTTP requests to HTTPS.
16. app.UseHttpsRedirection();
18. //This middleware is used to returns static files and short-circuits further request processing.
19. app.UseStaticFiles();
21. //This middleware is used to route requests.
22. app.UseRouting();
24. //This middleware is used to authorizes a user to access secure resources.
25. app.UseAuthorization();
27. //This middleware is used to add Razor Pages endpoints to the request pipeline.
28. app.UseEndpoints(endpoints =>
29. {
30. endpoints.MapRazorPages();
31. });
32. }

ASP.NET Core framework provides some of the Built-in middleware components that we can use easily add into the Configure method. Check out the [Microsoft documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1#built-in-middleware).

## Middleware Ordering

Middleware components are executed in the order they are added to the pipeline and care should be taken to add the middleware in the right order otherwise the application may not function as expected. This ordering is critical for security, performance, and functionality.

The following middleware components are for common app scenarios in the recommended order:



The first configured middleware has received the request, modify it (if required), and passes control to the next middleware. Similarly, the first middleware is executed at the last while processing a response if the echo comes back down the tube. That’s why Exception-handling delegates need to be called early in the pipeline, so they can validate the result and displays a possible exception in a browser and client-friendly way.

## Understanding the Run, Use and Map Method

**app.Run()**

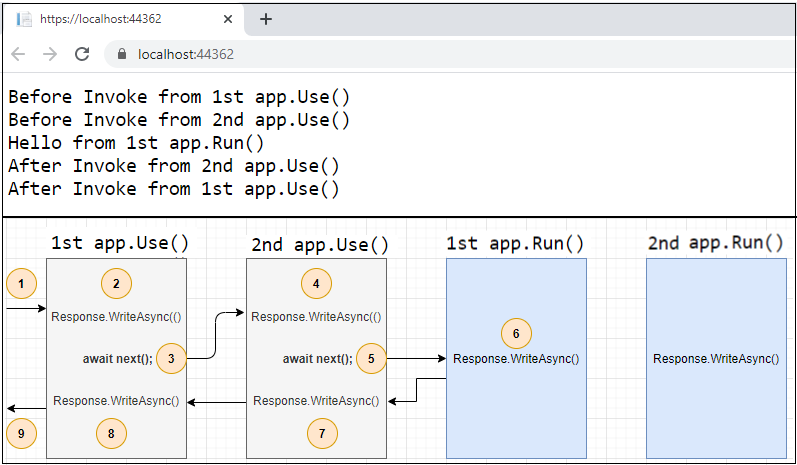
This middleware component may expose Run[Middleware] methods that are executed at the end of the pipeline. Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.

**app.Use()**

This is used to configure multiple middleware. Unlike app.Run(), We can include the next parameter into it, which calls the next request delegate in the pipeline. We can also short-circuit (terminate) the pipeline by not calling the next parameter.

Let’s consider the following example with the app.Use() and app.Run() and observe the output/response:

1. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
2. {
3. app.Use(async (context, next) =>
4. {
5. await context.Response.WriteAsync("Before Invoke from 1st app.Use()\n");
6. await next();
7. await context.Response.WriteAsync("After Invoke from 1st app.Use()\n");
8. });
10. app.Use(async (context, next) =>
11. {
12. await context.Response.WriteAsync("Before Invoke from 2nd app.Use()\n");
13. await next();
14. await context.Response.WriteAsync("After Invoke from 2nd app.Use()\n");
15. });
17. app.Run(async (context) =>
18. {
19. await context.Response.WriteAsync("Hello from 1st app.Run()\n");
20. });
22. // the following will never be executed
23. app.Run(async (context) =>
24. {
25. await context.Response.WriteAsync("Hello from 2nd app.Run()\n");
26. });
27. }



The first app.Run() delegate terminates the pipeline. In the following example, only the first delegate (“Hello from 1st app.Run()”) will run and the request will never reach the second Run method.

**app.Map()**

These extensions are used as a convention for branching the pipeline. The map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

Let’s consider the following example with the app.Map() and observe the output/response:

1. **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)
2. {
3. app.Map("/m1", HandleMapOne);
4. app.Map("/m2", appMap => {
5. appMap.Run(async context =>
6. {
7. await context.Response.WriteAsync("Hello from 2nd app.Map()");
8. });
9. });
10. app.Run(async (context) =>
11. {
12. await context.Response.WriteAsync("Hello from app.Run()");
13. });
14. }
15. **private** **static** **void** HandleMapOne(IApplicationBuilder app)
16. {
17. app.Run(async context =>
18. {
19. await context.Response.WriteAsync("Hello from 1st app.Map()");
20. });
21. }

The following table shows the requests and responses from localhost using the above code.

|  |  |
| --- | --- |
| **Request** | **Response** |
| https://localhost:44362/ | Hello from app.Run() |
| https://localhost:44362/m1 | Hello from 1st app.Map() |
| https://localhost:44362/m1/xyz | Hello from 1st app.Map() |
| https://localhost:44362/m2 | Hello from 2nd app.Map() |
| https://localhost:44362/m500 | Hello from app.Run() |

**Creating a Custom Middleware**

Middleware is generally encapsulated in a class and exposed with an extension method. The custom middleware can be built with a class with InvokeAsync() method and RequestDelegate type parameter in the constructor. RequestDelegate type is required in order to execute the next middleware in a sequence.

Let’s consider an example where we need to create custom middleware to log a request URL in a web application.

1. **public** **class** LogURLMiddleware
2. {
3. **private** **readonly** RequestDelegate \_next;
4. **private** **readonly** ILogger<LogURLMiddleware> \_logger;
5. **public** LogURLMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)
6. {
7. \_next = next;
8. \_logger = loggerFactory?.CreateLogger<LogURLMiddleware>() ??
9. **throw** **new** ArgumentNullException(nameof(loggerFactory));
10. }
11. **public** async Task InvokeAsync(HttpContext context)
12. {
13. \_logger.LogInformation($"Request URL: {Microsoft.AspNetCore.Http.Extensions.UriHelper.GetDisplayUrl(context.Request)}");
14. await **this**.\_next(context);
15. }
16. }
17. **public** **static** **class** LogURLMiddlewareExtensions
18. {
19. **public** **static** IApplicationBuilder UseLogUrl(**this** IApplicationBuilder app)
20. {
21. **return** app.UseMiddleware<LogURLMiddleware>();
22. }
23. }

In Configure method:

1. app.UseLogUrl();

## Enabling directory browsing through Middleware

Directory browsing allows users of your web app to see a directory listing and files within a specified directory.

Directory browsing is disabled by default for security reasons.

Let’s consider an example where we want to allow the list of images in the browser under the images folder in wwwroot. UseDirectoryBrowser middleware can handle and serve those images for that request and then short-circuit the rest of the pipeline.

1. app.UseDirectoryBrowser(**new** DirectoryBrowserOptions
2. {
3. FileProvider = **new** PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
4. RequestPath = "/images"
5. });

References:-

1. Geeks for Geeks.
2. Tutorial Point.
3. <https://www.w3schools.com>.
4. <https://docs.nopcommerce.com/en/developer/tutorials/description-of-xthe-web-config-file-in-project.html>.

5.https://www.c-sharpcorner.com.