**Dot Net Document:**

**1. What’s .Net**

**.NET** is a software framework that is designed and developed by Microsoft and .NET is an open-source and cross-platform development platform for building many types of applications like desktop, web, cloud, mobile, gaming, IoT, and AI apps. .Net framework supports more than 60 programming language to develop the software applications

**2.What’s ASP.Net**

ASP.NET is a set of libraries and tools to build web applications including front-end websites, APIs, and Microservices.

**3.What’s C#.Net:**

C# (C sharp): A modern object-oriented programming language that belongs to the C language family. C# enables developers to build many types of secure and robust applications that run in . NET.

**4.Output**

To output values or print text in C#, you can use the WriteLine() method:

**5.Comments**

Multi-line comments start with /\* and ends with \*/.

Single-line comments start with two forward slashes (//).

**6.Variables**

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

* int - stores integers (whole numbers), without decimals, such as 123 or -123
* double - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* string - stores text, such as "Hello World". String values are surrounded by double quotes
* bool - stores values with two states: true or false

**7.Data Types**

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable

**8.Type Casting**

Type casting is when you assign a value of one data type to another type.

**9.User Input**

The Console.ReadLine() method returns a string. Therefore, you cannot get information from another data type, such as int

**10.Operators**

Operators are used to perform operations on variables and values. Ex: +,-, \*, / , <=, >=, ==, &&, ||

**11.If...Else**

C# supports the usual logical conditions from mathematics:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false

**12.Methods**

A **method** is a block of code which only runs when it is called. You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

**13.Method Parameters**

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

static void MyMethod(string fname)

**14.Strings(Length, Upper, Lower, Concat, Interpolation, Substring, IndexOf, Escape character, +, Replace)**

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes

## **15.String Interpolation**

Another option of string concatenation, is **string interpolation**, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation

**16.Booleans**

A boolean type is declared with the bool keyword and can only take the values true or false

**17.Switch (default, goto case) – DEFINE DAYS**

Use the switch statement to select one of many code blocks to be executed.

* The switch expression is evaluated once
* The value of the expression is compared with the values of each case
* If there is a match, the associated block of code is executed
* The break and default keywords will be described later in this chapter

**18.Arrays(string arr, Int arr, foreach, sort, replace arr element, Linq – Min, max, sum)**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**

**19.For Loop**

Loop can be used to execute the code multiple times as how many times you want.

**20.Foreach:**

There is also a foreach loop, which is used exclusively to loop through elements in most of the collections and iteration can be defined based on the length of the collection object

**21.While/Do Loop**

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

The while loop loops through a block of code as long as a specified condition is True:

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

**22.Break/Continue**

It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**.

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

**23.Method Overloading**

With**method overloading**, multiple methods can have the same name with different parameters. We can also call this as compile time polymorphism.

**24.Classes/Objects**

A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit.

**25.Class Members**

Methods, Variables which are declared inside class

**26.Access Modifiers(Private, Public, Protected, Internal - Assemblies)**

Access modifier is used to define the scope of the variable

|  |  |
| --- | --- |
| public | The code is accessible for all classes |
| private | The code is only accessible within the same class |
| protected | The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about [inheritance](https://www.w3schools.com/cs/cs_inheritance.asp) in a later chapter |
| internal | The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter |

**27.Inheritence**

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in C# by which one class is allowed to inherit the features(fields and methods) of another class.

**28.Single:**

In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

**29.MultiLevel A, B, C:**

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C

**30.Multiple Inheritence:**

In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that **C# does not support multiple inheritance** with classes. In C#, we can achieve multiple inheritance only through Interfaces.

**31.Hybrid Inheritence A, B, ABC, ABD**

It is a mix of two or more of the above types of inheritance. Since C# doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In C#, we can achieve hybrid inheritance only through Interfaces.

**32.Hierarchical Inheritance - A, AB, AC**

 In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.

**33.Method overriding vs Method overloading**

Method Overriding in C# is similar to the [**virtual function in C++**](https://www.geeksforgeeks.org/virtual-function-cpp/). Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

**34.Base keyword**

This is used to access members of the base class from derived class. It basically used to access constructors and methods or functions of the base class. The base keyword cannot use within a static method. Base keyword specifies which constructor of the base class should be invoked while creating the instances of the derived class.

**35.Assemblies(Private, Public, Satellite – Resource):**

An assembly is **a file that is automatically generated by the compiler upon successful compilation of every .** **NET application**. It can be either a Dynamic Link Library or an executable file(dll or exe)

**36.Private Assembly**

A private assembly can be used by only a single application and not accessible for other applications. The dll or exe is generally stored in the application root folder.

**37.Public / Shared Assembly**

Public / Shared Assemblies are the assemblies that are accessible globally/shared across the machine to all the applications which are stored inside Global Assembly Cache (GAC), these can generally be found in the operating system folder like follows C:WindowsAssembly. For using the shared assemblies you need to register the assembly with a strong name in the GAC using gacutil.exe.

most common assemblies contained in the Global Assembly Cache,

**38.Satellite Assembly**

Satellite assemblies are useful while creating multilingual applications. Using atellite assemblies, localizable resources can be placed for different languages in different assemblies.

**Classes:  
39.Abstract class**

Abstraction in C# is the process to hide the internal details and show only the functionality. The abstract modifier indicates the incomplete implementation. The keyword abstract is used before the class or method to declare the class or method as abstract. Also, the abstract modifier can be used with indexers, events, and properties.

**40.Sealed class – Can’t inherit for security purpose**

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.

**41.Static class – This, can have static and non-static methods**

In C#, one is allowed to create a static class, by using *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**](https://www.geeksforgeeks.org/c-sealed-class/), means ***you cannot inherit a static class from another class***.

**42.Partial class – Logic can be splitted then can be added as a single class**

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled. A partial class is created by using a ***partial***keyword. This keyword is also useful to split the functionality of methods, interfaces, or structure into multiple files.

**43.Abstraction**

Abstraction is hiding the implementation of the methods/class to the consumer and provide the needed information only to the consumer.

**44.Encapsulation**

Encapsulation is data hiding(information hiding) and it can be achieved with the help of access modifiers.

**45.Properties**

A property is like a combination of a variable and a method, and it has two methods: a get and a set method

The get method returns the value of the variable name.

The set method assigns a value to the name variable.

**46.Polymorphism.**

Polymorphism, in C#, is the ability of objects of different types to provide a unique interface for different implementations of methods.

Early binding is the compile time Polymorphism also it can be referred as method overloading. Late binding is the run time Polymorphism and it can be referred as method Overriding

**47.Interface:**

An interface has the only method signatures and definition (with empty bodies). Interfaces cannot be used to create objects. It’s used to achieve security - hide certain details and only show the important details of an object (interface).

**48.Constructors**

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created. Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation. It is used to assign initial values to the data members of the same class. 

**49.Default Constructor**

A constructor with no parameters is called a default constructor. A default constructor has every instance of the class to be initialized to the same values. The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class

**50.Parameterized Constructor**

A constructor having at least one parameter is called as parameterized constructor. It can initialize each instance of the class to different values.

**51.Copy Constructor**

This constructor creates an object by copying variables from another object. Its main use is to initialize a new instance to the values of an existing instance.

**52.Private Constructor**

If a constructor is created with private specifier is known as Private Constructor. It is not possible for other classes to derive from this class and also it’s not possible to create an instance of this class.

**53.Static Constructor**

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

**54.Model POCO**

A Plain Old CLR Objects (POCO) is a class, which doesn't depend on any framework-specific base class. It is like any other normal .NET class. Due to this, they are called Plain Old CLR Objects.

**55.Value type**

A data type is a value type if it holds a data value within its own memory space. It means the variables of these data types directly contain values.

**56.Reference type**

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

**57.Collections**

Collections standardize the way of which the objects are handled by your program. In other words, it contains a set of classes to contain elements in a generalized manner. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc

**58.Array**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

**58.Dictionary**

Dictionary is a generic collection which is generally used to store key/value pairs. The working of Dictionary is quite similar to the [non-generic hashtable](https://www.geeksforgeeks.org/c-sharp-hashtable-with-examples/).

The advantage of Dictionary is, it is generic type. Dictionary is defined under System.Collection.Generic namespace. It is dynamic in nature means the size of the dictionary is grows according to the need.

**59.List**

**List class** represents the list of objects which can be accessed by index. It comes under the **System.Collection.Generic** namespace. List class can be used to create a collection of different types like integers, strings etc. List<T> class also provides the methods to search, sort, and manipulate lists.

**60.SortedList**

SortedList class is a collection of **(key, value)** pairs which are sorted according to keys. Those pairs can be accessible by key and as well as by index(zero-based indexing). This comes under **System.Collections** namespace.

**61.Enums – To maintain the constant values  
Enumeration (or enum)** is a [value data type](https://www.geeksforgeeks.org/c-data-types-2/) in C#. It is mainly used to assign the names or string values to integral constants, that make a program easy to read and maintain

**62.Files – Create, Write, Delete**

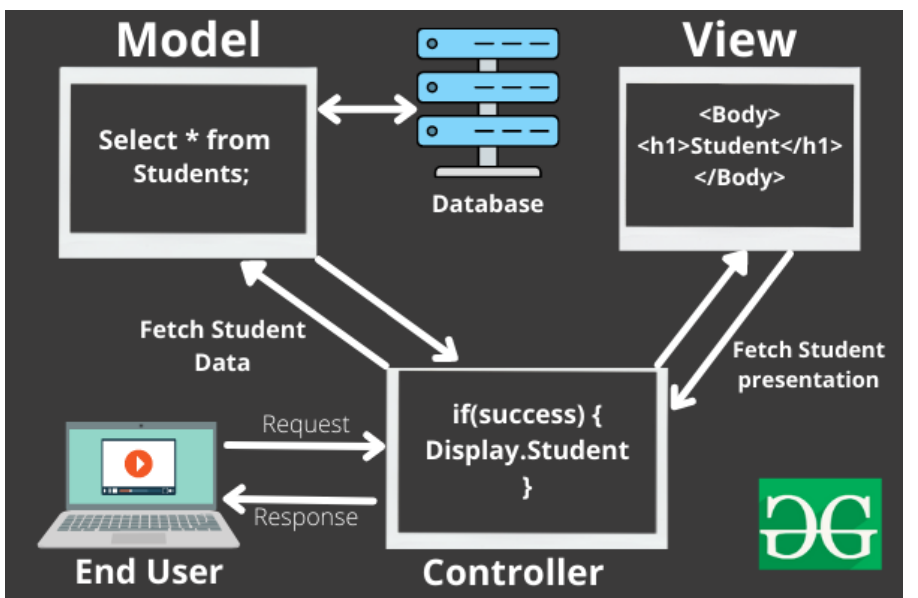
Generally, the file is used to store the data. The term File Handling refers to the various operations like creating the file, reading from the file, writing to the file, appending the file, etc. There are two basic operation which is mostly used in file handling is reading and writing of the file. The file becomes stream when we open the file for writing and reading. A stream is a sequence of bytes which is used for communication

**63.Exceptions – throw new exception**

An exception is defined as an event that occurs during the execution of a program that is unexpected by the program code. The actions to be performed in case of occurrence of an exception is not known to the program. In such a case, we create an exception object and call the exception handler code. The execution of an exception handler so that the program code does not crash is called exception handling.

**MVC Framework Introduction**

The [**Model-View-Controller (MVC)**](https://www.geeksforgeeks.org/mvc-design-pattern/) framework is an architectural/design pattern that separates an application into three main logical components **Model**, **View**, and **Controller**. Each architectural component is built to handle specific development aspects of an application



**Features of MVC :**

* It provides a **clear separation** of business logic, Ul logic, and input logic.
* It supports **Test Driven Development (TDD).**
* It offers full control over your HTML and URLs which makes it easy to design web application architecture.

**Disadvantages of MVC:**

* Increased complexity and Inefficiency of data
* It is not suitable for building small applications.

**MVC Fundamentals:**

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute();

app.MapRazorPages();

**Controller:**

The controller is the component that enables the interconnection between the views and the model so it acts as an intermediary.  It process all the business logic and incoming requests, manipulate data using the **Model**component and interact with the **View**to render the final output.

**View:**

The **View**component is used for all the UI logic of the application. It generates a user interface for the user. Views are created by the data which is collected by the model component but these data aren’t taken directly but through the controller.

**Model:**

The **Model**component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data.

**Razer View:**

Razor View engine is a markup syntax which helps us to write HTML and server-side code in web pages using C# or VB.NET.

Razor is a templating engine and ASP.NET MVC has implemented a view engine which allows us to use Razor inside of an MVC application to produce HTML. However, Razor does not have any ties with ASP.NET MVC.

**Explanation about IIS Express**

@{

var price = 101;

}

@{

if(price == 100)

{

<p>It's hundred</p>

}

else

{

<p>It's not hundred</p>

}

}

**ASP.NET MVC Core Action Methods**

Every public method of the controller is ActionMethod except methods marked as [NonAction]. Action methods are similar to normal methods however ActionMethod has limitations as

* Action method must be public, it can not be private.
* Action method can not be static or extension method.
* Action method can not be getter or setter.

[ActionName("GetProductNameByID")]

public string GetProductName(int ProductID)

{

return "Product name is ABC.";

}

[ActionName("GetProductNameByProductCode")]

public string GetProductName(string code)

{

return "Product name is ABC.";

}

**ActionResult**

MVC framework includes various Result classes, which can be returned from an action method. The result classes represent different types of responses, such as HTML, file, string, JSON, javascript, etc. The following table lists all the result classes available in ASP.NET MVC.

| Result Class | Description |
| --- | --- |
| ViewResult | Represents HTML and markup. |
| EmptyResult | Represents No response. |
| ContentResult | Represents string literal. |
| FileContentResult/ FilePathResult/ FileStreamResult | Represents the content of a file. |
| JavaScriptResult | Represent a JavaScript script. |
| JsonResult | Represent JSON that can be used in AJAX. |
| RedirectResult | Represents a redirection to a new URL. |
| RedirectToRouteResult | Represent another action of same or other controller. |
| PartialViewResult | Returns HTML from Partial view. |
| HttpUnauthorizedResult | Returns HTTP 403 status. |

**Routing in ASP.NET Core MVC**

Routing is the process through which the application matches an incoming URL path and executes the corresponding action methods. ASP.NET Core MVC uses a routing middleware to match the URLs of incoming requests and map them to specific action methods.

There are two types of routing for action methods:

* [Conventional Routing](https://code-maze.com/routing-asp-net-core-mvc/#conventionalrouting)
* [Attribute Routing](https://code-maze.com/routing-asp-net-core-mvc/#attributerouting)

**Conventional Routing**

Configure routing in Configure method in startup.cs class

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

endpoints.MapControllerRoute(

name : "employee",

pattern: "EmployeeDetails",

defaults: new { controller = "Employee", action = "Index" });

});

**Attribute Routing**

* By placing a route on the controller or the action method, we can make use of the Attribute Routing feature.
* Let’s modify the Configure() method in the startup.cs class and remove the default routes that we had defined earlier.

[Route("[controller]/[action]")]

[Route("[controller]")]

public class TestController : Controller

{

[Route("")] // Matches 'Test'

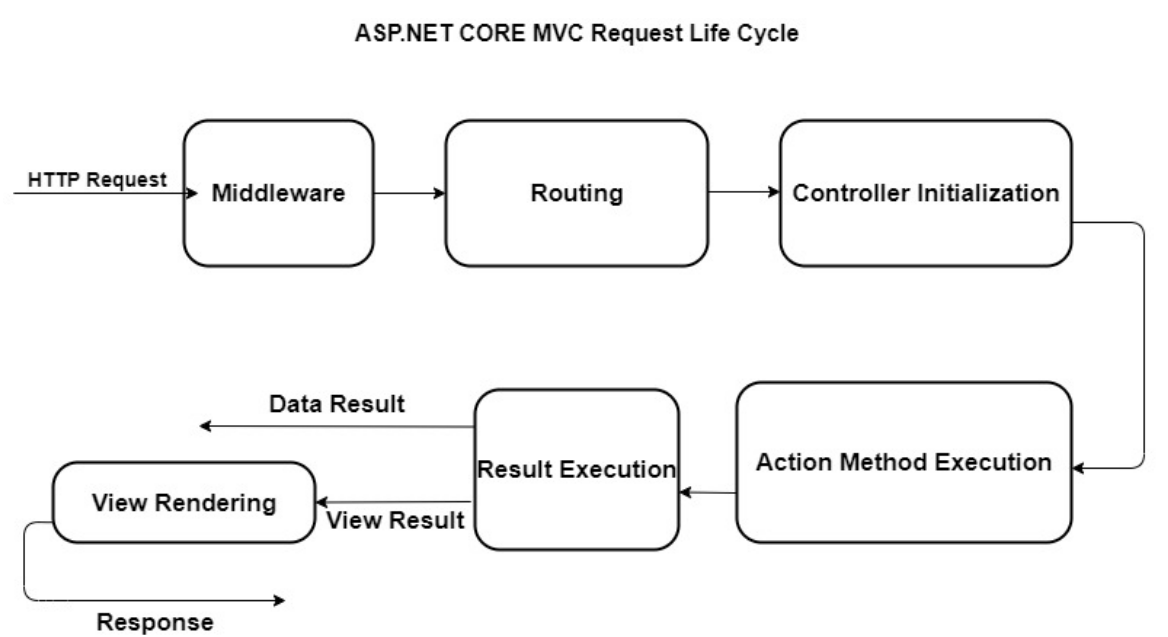
[Route("Index")] // Matches 'Test/Index'

public IActionResult Index()

}

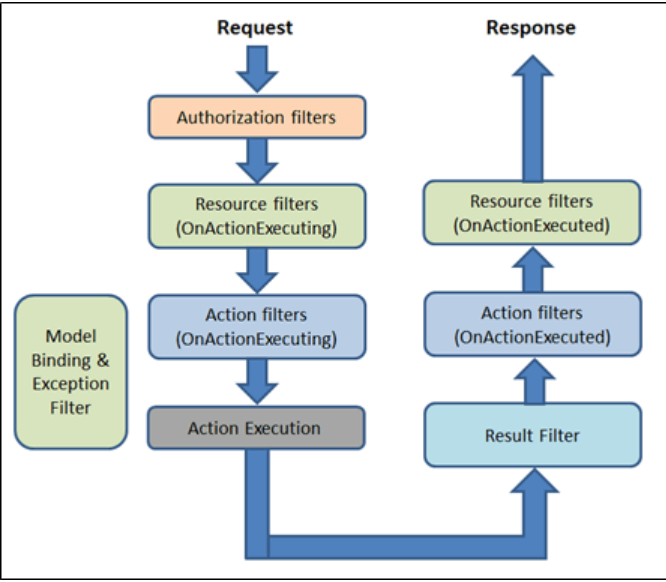
**ASP.NET Core MVC Request Life Cycle/Pipeline**

The ASP.NET Core MVC Request Life Cycle is a sequence of events, stages or components that interact with each other to process an HTTP request and generate a response that goes back to the client. In this article, we will discuss each and every stage of ASP.NET Core MVC Request Life Cycle in detail.



**Filters in MVC**

Every filter type is executed at a different stage in the filter pipeline. Following are the filter types.



***Authorization filters***  
The Authorization filters are executed first. This filter helps us to determine whether the user is authorized for the current request. It can short-circuit a pipeline if a user is unauthorized for the current request. We can also create custom authorization filter.

public class AuthorizeActionFilter : Attribute, IAuthorizationFilter

{

public void OnAuthorization(AuthorizationFilterContext context)

{

//Write you code here to authorize or unauthorize the user

}

}

***Resource filters***

The Resource filters handle the request after authorization. It can run the code before and after the rest of the filter is executed. This executes before the model binding happens. It can be used to implement caching.

public class CustomResourceFilterAttribute : Attribute, IResourceFilter

{

public void OnResourceExecuting(ResourceExecutingContext context)

{

context.Result = new ContentResult()

{

Content = "This is a Resource filter."

};

}

public void OnResourceExecuted(ResourceExecutedContext context)

{

}

}

***Action filters***  
The Action filters run the code immediately before and after the controller action method is called. It can be used to perform any action before or after execution of the controller action method. We can also manipulate the arguments passed into an action.

public class CustomActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

// Executed before execution of an action method

}

public void OnActionExecuted(ActionExecutedContext context)

{

// Executed after execution of an action method

}

}

***Exception filters***  
The Exception filters are used to handle exception that occurred before anything written to the response body.

public class CustomExceptionFilter : Attribute, IExceptionFilter

{

public void OnException(ExceptionContext context)

{

context.Result = new ViewResult()

{

StatusCode = (int)HttpStatusCode.BadRequest,

ViewName = "Error"

};

context.ExceptionHandled = true;

}

}

***Result filters***  
The Result filters are used to run code before or after the execution of controller action results. They are executed only if the controller action method has been executed successfully.

public class CustomResultFilter : Attribute, IResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

context.Result = new ViewResult

{

ViewName = "Hello"

};

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

**Data Annotation**

**Data Annotations** are nothing but certain validations that we put in our models to validate the input from the user. ASP.NET MVC provides a unique feature in which we can validate the models using the Data Annotation attribute. Import the following namespace to use data annotations in the application.

System.ComponentModel.DataAnnotations

It is very easy to use and the code becomes much cleaner as compared to normal ASP.NET validators.

Let us understand some of the validator attributes that we can use in MVC.

## **Types of Data Annotations in ASP.NET MVC**

### **Required**

This attribute specifies that the value is mandatory and cannot be skipped.

**Syntax**

[Required(ErrorMessage="Please enter name"),MaxLength(30)]

### **DataType**

This attribute is used to specify the datatype of the model.

**Syntax**

[DataType(DataType.Text)]

### **Range**

Using this attribute we can set a range between two numbers.

**Syntax**

[Range(100,500,ErrorMessage="Please enter correct value")]

### **StringLength**

Using this attribute we can specify maximum and minimum length of the property.

**Syntax**

[StringLength(30,ErrorMessage="Do not enter more than 30 characters")]

### **DisplayName**

Using this attribute we can specify property name to be displayed on view.

**Syntax**

[Display(Name="Student Name")]

### **MaxLength**

Using this attribute we can specify maximum length of property.

**Syntax**

[MaxLength(3)]

### **Bind**

This attribute specifies fields to include or exclude for model binding.

**Syntax**

[Bind(Exclude = "StudentID")]

### **DisplayFormat**

This attribute allows us to set date in the format specified as per the attribute.

**Syntax**

[DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}")]

### **RegularExpression**

We can set a regex pattern for the property. For ex: Email ID.

**Syntax**

[RegularExpression(@"^\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$", ErrorMessage = "Email is not valid.")]

**Ex:**

public class UserRegistration

{

[Required(ErrorMessage = "Please enter your first name")]

[MinLength(2, ErrorMessage = "Please enter atleast two characters")]

[MaxLength(50, ErrorMessage = "Please enter upto 50 characters")]

[Display(Name = "First Name")]

public string FirstName { get; set; }

[Required(ErrorMessage = "Please enter your last name")]

[MinLength(2, ErrorMessage = "Please enter atleast two characters")]

[MaxLength(50, ErrorMessage = "Please enter upto 50 characters")]

[Display(Name = "Last Name")]

public string LastName { get; set; }

[Required(ErrorMessage = "Please Enter your age")]

[DataType(DataType.PostalCode)]

[MaxLength(3, ErrorMessage = "Please enter valid age")]

[Display(Name = "Age")]

public int Age { get; set; }

[Required(ErrorMessage = "Please provide your email id")]

[MinLength(10, ErrorMessage = "Please enter atleast two characters")]

[MaxLength(70, ErrorMessage = "Please enter upto 50 characters")]

[DataType(DataType.EmailAddress)]

[Display(Name = "Email Address")]

public string EmailId { get; set; }

[Required(ErrorMessage = "Please provide your mobile number")]

[MaxLength(10, ErrorMessage = "Please enter upto 10 digits")]

[DataType(DataType.PhoneNumber)]

[Display(Name = "Mobile Number")]

public long MobileNumber { get; set; }

}

**Strongly typed view**

The view which binds to a specific type of ViewModel is called as **Strongly Typed View**. By specifying the model, the Visual studio provides the intellisense and compile time checking of type.

**Advantages of Strongly Typed View**

1. IntelliSense Help
2. Compile time error checking
3. You do not have to cast between types

*Since there is only one Model Property, you can have only one ViewModel per View.*

**Ex:**

@model UserManagement.Models.UserRegistration

@{

ViewData["Title"] = "UserRegistration";

}

<style>

.custom-width

{

width: 600px;

background-color : red;

}

</style>

<h4>UserRegistration</h4>

<hr />

<div class="row">

<div class="col-md-4">

<**form** **asp-action**="UserRegistration">

<**div** **asp-validation-summary**="ModelOnly" class="text-danger"></**div**>

<div class="form-group">

<**label** **asp-for**="FirstName" class="control-label"></**label**>

<**input** **asp-for**="FirstName" class="form-control custom-width" />

<**span** **asp-validation-for**="FirstName" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="LastName" class="control-label"></**label**>

<**input** **asp-for**="LastName" class="form-control" />

<**span** **asp-validation-for**="LastName" class="text-danger"></**span**>

</div>

<div class="form-group">

<**label** **asp-for**="Age" class="control-label"></**label**>

<**input** **asp-for**="Age" class="form-control" />

<**span** **asp-validation-for**="Age" class="text-danger"></**span**>

</div>

**ASP.NET MVC Scaffolding**

Scaffolding is used to define the code-generation framework used in web applications. It uses T4 templates to generate basic controllers and views for the models. It generates instances for the mapped domain model and code for all CRUD operations. It also reduces the amount of time for developing a standard data operation in the application.

Basically, it is an automated code generation framework, it generates code for CRUD operations based on the provided domain model classes

**Transport data from controller to view**

@model Employee

<h2>Employee Detail:</h2>

<ul>

<li>Student Id: @Model.EmployeeId</li>

<li>Student Name: @Model. EmployeeName</li>

<li>Age: @Model.Age</li>

</ul>

**Authentication And Authorization In ASP.NET Core MVC**

https://www.c-sharpcorner.com/article/authentication-and-authorization-in-asp-net-core-mvc-using-cookie/#:~:text=Authentication%20And%20Authorization%20In%20ASP.NET%20Core%20MVC%20Using%20Cookie,-Mukesh%20Kumar&text=Security%20is%20the%20main%20concern,think%20about%20Authentication%20and%20Authorization.