**Data Annotation**

Data annotations are attributes that can be applied to a class, property, or method to provide additional information about that element.

Some common data annotations in C# include:

* Required: Specifies that a property is required and cannot be null or empty.
* StringLength: Specifies the minimum and maximum length of a string property.
* Range: Specifies the minimum and maximum value of a numeric property.
* EmailAddress: Specifies that a string property must be a valid email address.
* RegularExpression: Specifies a regular expression that a string property must match.

Example:

public class Person

{

[Required(ErrorMessage = "Name is required")]

public string Name { get; set; }

[Range(18, 30, ErrorMessage = "Age must be between 18 and 30")]

public int Age { get; set; }

[EmailAddress(ErrorMessage = "Invalid email address")]

public string Email { get; set; }

}

**Exception Filter**

An exception filter is a mechanism in some programming languages, such as C#, that allows developers to catch and handle specific exceptions based on custom criteria.

When an exception is thrown in a try block, the runtime checks each catch block in turn to see if the thrown exception matches the exception type specified in the catch block. If the exception type matches, the corresponding catch block is executed.

However, with an exception filter, you can further refine which exceptions are caught by specifying additional conditions that the exception must meet. The filter is defined using the when keyword followed by a Boolean expression. If the expression evaluates to true, the corresponding catch block is executed; otherwise, the next catch block is checked.

Example:

using System;

class Program

{

static void Main(string[] args)

{

try

{

throw new ArgumentException("Invalid argument value");

}

catch (Exception ex) when (ex is ArgumentException && ((ArgumentException)ex).ParamName == "value")

{

Console.WriteLine("Caught ArgumentException with parameter name 'value'");

}

catch (Exception ex)

{

Console.WriteLine("Caught exception: " + ex.Message);

}

}

}

**Action Filter**

Action filters in web development are used to perform certain actions before or after executing an action method in an MVC or API controller.

They are a type of filter that can be applied to one or more action methods in a controller. The main purpose of action filters is to modify or inspect

the HTTP request,the action method, or the HTTP response that is returned by the action method.

Example:

/\*Separate class where Custom action filter is defined\*/

public class CustomActionFilter : IActionFilter

{

private readonly ILogger \_logger;

public CustomActionFilter(ILogger<CustomActionFilter> logger)

{

\_logger = logger;

}

public void OnActionExecuting(ActionExecutingContext context)

{

\_logger.LogInformation("Executing action filter before method called");

}

public void OnActionExecuted(ActionExecutedContext context)

{

\_logger.LogInformation("Executing action filter after method called");

}

}

/\* how action filter is applied to specific method\*/

[TypeFilter(typeof(CustomActionFilter))]

public IEnumerable<Employee\_Registration> getdetails()

{

\_logger.LogInformation("Executed Inside Get method.");

return \_employeeInterfaces.getdetails();

}

/\*Assigning logs to Console\*/

public void ConfigureServices(IServiceCollection services)

{

services.AddLogging(builder =>

{

builder.AddConsole();

});

services.AddScoped<CustomActionFilter>();

}

**Inversion of Control**

Inversion of Control (IoC) is a design principle in software development that refers to the reversal of the traditional flow of control in a program. In traditional programming, a program's flow of control is determined by the code that calls other components or services. In contrast, in an IoC architecture, the control flow is inverted, and components or services rely on a container or framework to provide them with the necessary dependencies and services.

One of the most common implementations of IoC is through Dependency Injection (DI), which is a pattern for providing objects with their required dependencies. In DI, instead of creating an object and its dependencies within its code, the object's dependencies are passed in from an external source, typically a container or framework. This allows the object to be more modular and easier to test, as the dependencies can be easily mocked or replaced.

Example:

/\*Dependancy Injection\*/

public class AuthenticationController : ControllerBase

{

private readonly IEmployeeInterfaces \_employeeInterfaces;

public AuthenticationController(IEmployeeInterfaces employeeInterfaces)

{

\_employeeInterfaces = employeeInterfaces;

}

/\*Demonstration of Dependancy Injection\*/

public IEnumerable<Employee\_Registration> getdetails()

{

\_logger.LogInformation("Executed Inside Get method.");

return \_employeeInterfaces.getdetails();

}