**Data annotation:**

* Data annotation is a feature in c# programming language that allows developers to specify additional metadata about types, properties and methods, which can be used by other programs or libraries to perform certain actions or provide additional information.
* We can easily add validation to our application by adding Data Annotations to our model classes.
* Data Annotations allow us to describe the rules we want applied to our model properties, and ASP.NET MVC will take care of enforcing them and displaying appropriate messages to our users.

Ex:

public class Person

{

[Required]

public string Name { get; set; }

[Range(18, 60)]

public int Age { get; set; }

[EmailAddress]

public string Email { get; set; }

}

In the above example, we have used the Required attribute to indicate that the Name property is required, the Range attribute to specify that the Age property should be between 18 and 60, and the EmailAddress attribute to validate that the Email property is a valid email address.

**Inversion of Control:**

* Inversion of Control (IoC) is a programming design pattern, that describes the inversion of the flow of control in a software application.
* Inversion of Control (IoC) is a design principle where the control of object creation and its lifecycle is inverted or moved outside of the class that depends on it.
* Instead of an object creating its dependencies, the dependencies are injected into the object when it is created.
* Both inversion of control and dependency injection enable us to break dependencies between the components in our application and make our application easier to test and maintain.
* However, inversion of control and dependency injection are not the same — there are subtle differences between the two.

Ex: public class ProductService  
    {  
        private readonly FileLogger \_fileLogger = new FileLogger();  
        public void Log(string message)  
        {  
            \_fileLogger.Log(message);  
        }  
    }

public class FileLogger  
    {  
        public void Log(string message)  
        {  
            Console.WriteLine("Inside Log method of FileLogger.");  
            LogToFile(message);  
        }

**Action Filter:**

* Action filters are attributes in ASP.NET MVC that allow you to apply additional processing to an action method before or after execution.
* They provide a way to customize the behavior of an action method or the response returned by the action method.
* This filter is really helpful when we have to perform some pre or post-processing logic with action method executed.
* Action filter gets called before an action method gets executes and after an action method get executes.

Ex:

[LogActionFilter]

public ActionResult Index()

{

// Perform some action

return View();

}

This will cause the LogActionFilter to execute before and after the Index action method, allowing us to log the details of the request and response.

The contents of an action filter depend on its specific implementation and purpose. However, in general, an action filter can contain the following:

**1. Code to execute before or after an action method is executed**: This can include performing validation, logging, or other tasks.

**2. Data that is passed between the action filter and the action method:** This can be used to pass information about the request or response, or to modify the behavior of the action method.

**3. Logic to modify the result returned by the action method:** This can include modifying the view or redirecting the user to a different page.

**4. Error handling logic:** This can include handling exceptions that occur during the execution of the action method.

**5. Authentication and authorization logic**: This can include checking the user's credentials or permissions before allowing them to access the action method.

**6. Configuration settings:** This can include settings that control the behavior of the action filter, such as the order in which it is executed or the conditions under which it should be applied.

**Exception Filter:**

* Exception filters are used in ASP.NET MVC applications to handle unhandled exceptions that may occur during the execution of an action method.
* An exception filter is a feature in ASP.NET Web API, that allows us to catch and handle exceptions that occur during the execution of a Web API action.
* It gives us the ability to create a central location for handling errors that occur during the execution of a Web API application.
* Exception filters are useful because they allow us to provide a consistent error handling mechanism for all actions in a Web API application.
* Exception filters are executed when an unhandled exception occurs during the execution of a Web API action.

Ex:

using System.Web.Mvc;

public class CustomExceptionFilter : FilterAttribute, IExceptionFilter

{

public void OnException(ExceptionContext filterContext)

{

if (!filterContext.ExceptionHandled)

{

filterContext.ExceptionHandled = true;

filterContext.Result = new ViewResult

{

ViewName = "Error",

ViewData = new ViewDataDictionary(filterContext.Exception)

};

} } }

[CustomExceptionFilter]

public class HomeController : Controller

{

public ActionResult Index()

{

throw new Exception("Something went wrong");

}

}

**Error Models:**

* Error models are a way to handle errors and exceptions in software development. In a typical web application, when an error occurs, it can be difficult to trace the root cause of the problem.
* Error models are a way to organize and standardize error handling and reporting, making it easier to find and fix errors.

A basic example of an error model could be a class that contains properties for the error message, error code, and stack trace. For instance, in C#, the class could be defined as follows:

Ex: public class ErrorModel

{

public string ErrorMessage { get; set; }

public int ErrorCode { get; set; }

public string StackTrace { get; set; }

}

Whenever an error occurs in the application, an instance of this class can be created and populated with the relevant information, such as the error message and stack trace.

This error model can then be logged, sent to an error reporting service, or returned to the client in a standardized format.

By using a consistent error model throughout the application, developers can more easily understand and handle errors as they arise, leading to a more stable and reliable application.

**Life cycle of Data Annotation:**

The life cycle of data annotation can be summarized as follows:

1. **Model Creation:** At the start of the life cycle, the model is created with its properties and attributes.
2. **Annotation:** Annotations are added to the properties and classes of the model. These annotations define validation rules, display formats, and other metadata.
3. **Model Binding:** When a request is received by the application, the data is bound to the model. The data binding process applies the validation rules defined in the annotations and checks the validity of the data.
4. **Validation:** After the data is bound to the model, the validation process is initiated. The validation process checks the data against the validation rules defined in the annotations and generates error messages if the data is not valid.
5. **Error Handling:** If validation fails, the error messages are generated and displayed to the user. Error handling may involve displaying error messages, logging errors, or returning error responses to the client.
6. **Rendering:** If the data is valid, the data can be rendered in a format specified by the annotations. This may involve converting data types, formatting data, or generating user interfaces.
7. **Update:** If the user modifies the data, the updated data is bound to the model and the validation and rendering process are repeated.

**Request Life Cycle:**

* The request life cycle refers to the sequence of events that take place when a client sends a request to a server and the server responds with a corresponding response.
* In the context of web development, this typically involves a client sending a request to a web server through a web browser, and the server responding with HTML, CSS, and JavaScript that the browser can render.

The request life cycle typically involves several stages:

1. **Client sends a request to the server:** The client (usually a web browser) sends a request to the server over the network. The request typically includes a URL, HTTP method (such as GET, POST, PUT, DELETE, etc.), headers, and any payload data.
2. **Server processes the request:** The server receives the request, parses it, and performs any necessary processing. This may involve validating the request, routing it to the appropriate controller or endpoint, and retrieving or modifying data in a database or other data source.
3. **Server generates a response:** Once the server has processed the request, it generates a response that includes an HTTP status code (such as 200 OK or 404 Not Found), headers, and any payload data (such as HTML, JSON, XML, or binary data).
4. **Server sends the response to the client:** The server sends the response back to the client over the network. The client then receives the response, parses it, and renders it as appropriate (for example, by displaying HTML in a web browser).
5. **Client may send additional requests:** The client may send additional requests to the server, depending on the user's actions and the application's logic.

The request life cycle is a fundamental concept in web development and it can help developers to optimize their applications for performance, security, and scalability.