**ASP DOT NET MVC**

**✅ 1. What is .NET Framework?**

The **.NET Framework** is a software development platform made by Microsoft. It helps developers build and run apps, especially on Windows.

**🔑 Key Points:**

* It provides a **runtime** (called CLR – Common Language Runtime) that manages code execution.
* It includes a large **library** of reusable code (called the Framework Class Library or FCL).
* Developers can use languages like **C#, VB.NET, F#**, etc.
* You can create **Windows apps, web apps, console apps, web services**, and more.

**Example:** When you write code in C#, the .NET Framework helps convert it to something the computer understands and manages memory, errors, etc.

**✅ 2. .NET Architecture**

The architecture of .NET Framework includes several layers that work together:

**🧱 Main Components:**

1. **CLR (Common Language Runtime)**:
   * Heart of the .NET Framework.
   * Manages memory, security, exceptions, and more.
   * Converts your C#/VB.NET code into **MSIL** (Microsoft Intermediate Language) and then to **machine code**.
2. **FCL (Framework Class Library)**:
   * Collection of pre-written code to handle common programming tasks.
   * Includes classes for file handling, database access, web requests, etc.
3. **Languages (C#, VB.NET)**:
   * You write code using a language like C#.
   * CLR supports multiple languages.
4. **ASP.NET**:
   * A part of the .NET Framework for building **web applications**.

**💡 Flow:**

Your C# code → CLR compiles it to MSIL → JIT Compiler → Machine Code → Runs on your system

**✅ 3. What is a Server?**

A **server** is a computer or system that provides data or services to other computers (called **clients**).

**🔧 Examples:**

* When you visit a website, your browser (client) sends a request to a **web server**.
* The server processes the request and sends back the webpage.

**🖥️ Types of Servers:**

* **Web Server**: Serves websites (like IIS, Apache)
* **Database Server**: Stores and manages data (like SQL Server)
* **Mail Server**: Handles email

**✅ 4. What is XAMPP?**

**XAMPP** is a free software package that helps developers run and test web applications on their own computer.

**🌐 It includes:**

* **X** – Cross-platform
* **A** – Apache (Web Server)
* **M** – MySQL (Database)
* **P** – PHP
* **P** – Perl

Even though XAMPP is mostly used for PHP development, it's good to know it if you're comparing with ASP.NET.

**🔎 Why Use XAMPP?**

* Lets you run a server environment **locally**.
* Good for testing websites before uploading online.

**✅ 5. What is MVC Structure?**

**MVC** stands for **Model – View – Controller**. It’s a **design pattern** used in ASP.NET to organize code in a better way.

**🧩 Breakdown:**

1. **Model**:
   * Deals with **data and business logic**.
   * Example: Saving user info, calculations, etc.
2. **View**:
   * Deals with **UI (User Interface)** – what the user sees.
   * Example: HTML pages with data.
3. **Controller**:
   * Handles **user input** and controls the flow.
   * Takes user requests, interacts with the Model, and returns a View.

**💡 Real-Life Example:**

Think of a **food delivery app**:

* **Model** = Menu data, orders, payment logic
* **View** = The screen where you see the food menu
* **Controller** = When you click “Order,” it fetches the right food data and places the order

**✅ 6. What is a Framework?**

A **framework** is a **ready-made structure** that helps developers write programs without starting from scratch.

**🔧 Key Points:**

* Provides **reusable tools, libraries, and guidelines**.
* Makes coding **faster and more organized**.
* Helps reduce bugs and increases productivity.

**Example:**

ASP.NET is a **web development framework**. It gives you tools to make websites quickly without writing everything from zero.

**✅ 1. What is ASP.NET?**

**ASP.NET** is a **web development framework** made by Microsoft. It allows you to build **dynamic websites, web apps, and web services** using .NET technologies like C# and VB.NET.

**🔧 Key Features:**

* Works with the **.NET Framework** or **.NET Core/.NET 5+**
* Supports **MVC architecture**
* Allows creation of **HTML, CSS, and JavaScript-based pages**
* Secure and supports **session management, authentication, authorization**, etc.

**🖥️ Example:**

If you want to create a website like an online shop, you can use **ASP.NET** to build the backend (processing orders, storing data) and generate the frontend (HTML pages shown to users).

**✅ 2. What is Razor?**

**Razor** is a **syntax** used in ASP.NET for combining **HTML and C# code** in the same file.

**📄 Example:**

html

CopyEdit

<h1>Hello, @name!</h1>

If name = "Tishat", the output will be:

html

CopyEdit

Hello, Tishat!

**🔑 Key Points:**

* Starts C# code with @ symbol
* Used in **.cshtml** files
* Makes web pages **dynamic** by embedding server-side logic

**⚡ Why Use Razor?**

* Clean and easy to read
* No need for separate backend/frontend files
* Works seamlessly with MVC views

**✅ 4. Routing in MVC**

**Routing** is the system that decides **which controller and action** should handle a user's request.

**🔍 Example:**

If the user visits:

ruby

CopyEdit

https://yourapp.com/student/details/5

Routing will match this to:

csharp

CopyEdit

Controller: StudentController

Action: Details(int id) => id = 5

**💡 Default Route Pattern:**

csharp

CopyEdit

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

**⚙️ Key Terms:**

* **Controller**: Class that handles logic
* **Action**: Method inside a controller
* **id**: Optional parameter

**✅ 5. Life Cycle of an ASP.NET Page**

The **life cycle** is the series of steps an ASP.NET page goes through **from request to response**.

**🌀 Key Stages (Simplified):**

1. **Page Request**  
   Browser sends a request to the server.
2. **Start**  
   Page properties like Request, Response, User, etc. are set.
3. **Initialization**  
   Controls (like buttons, textboxes) are initialized with default values.
4. **Load**  
   Controls get actual data (e.g., from database or ViewBag/ViewData).
5. **Postback Event Handling**  
   If a user clicked a button or submitted a form, that event is processed here.
6. **Rendering**  
   ASP.NET converts page + controls into **HTML**.
7. **Unload**  
   Clean-up operations, closing DB connections, etc.

**🔁 Extra:**

* If the request is a **PostBack**, previous values are retained.
* You can write code in Page\_Load() to perform actions when the page is loaded.

**✅ ASP.NET vs ASP.NET Core**

|  |  |  |
| --- | --- | --- |
| **Feature/Topic** | **ASP.NET (Classic)** | **ASP.NET Core** |
| **Platform** | Runs **only on Windows** | Runs on **Windows, macOS, Linux (Cross-platform)** |
| **Release Year** | Introduced in **2002** | Released in **2016** |
| **Performance** | **Slower** compared to Core | **Faster**, lightweight |
| **Modular** | **Monolithic** (everything bundled together) | **Modular**, you only include what you need |
| **Web Server** | Uses **IIS** (Internet Information Services) | Can use **Kestrel** + IIS, Apache, or Nginx |
| **Development Model** | Based on **.NET Framework** | Based on **.NET Core / .NET 5+** |
| **Dependency Injection** | **Not built-in** by default | **Built-in** by design |
| **Razor Pages & Blazor** | Limited or Not Available | **Supported natively** |
| **Open Source** | Partially open-source | **Fully open-source** |
| **Support/Future** | **No new major updates**; legacy support only | **Actively developed and improved** |

**🔧 1. ASP.NET (Classic)**

ASP.NET is the **older version** built on the **.NET Framework**. It includes:

* **Web Forms**
* **MVC (ASP.NET MVC 5)**
* **Web API**

You’ll mostly use it if:

* You're working on **existing enterprise projects**
* You are **deploying on Windows servers**

**⚡ 2. ASP.NET Core**

ASP.NET Core is the **modern, fast, cross-platform** version of ASP.NET.

It supports:

* **MVC**
* **Razor Pages**
* **Blazor**
* **Minimal APIs**
* And more...

Use ASP.NET Core if:

* You're starting a **new project**
* You want **better performance**
* You want to run your app on **Linux, Mac, or Docker**

**🔍 Example Differences in Code:**

**ASP.NET MVC Controller (Old)**

csharp

CopyEdit

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

**ASP.NET Core Controller (New)**

csharp

CopyEdit

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

}

**Looks similar**, but ASP.NET Core has many **new features** under the hood!

**📢 Summary:**

🔹 **ASP.NET** = Old, Windows-only, suitable for legacy projects  
🔹 **ASP.NET Core** = New, fast, modern, cross-platform, and the future of .NET development

**✅ ASP.NET MVC (.NET Framework) Project Structure**

When you create a new **ASP.NET MVC project**, you'll see folders like this:

arduino

CopyEdit

YourProjectName/

│

├── Controllers/

├── Models/

├── Views/

├── App\_Start/

├── Scripts/

├── Content/

├── App\_Data/

├── Global.asax

├── Web.config

Let’s break down each one 👇

**📁 1. Controllers Folder**

* Contains **controller classes** — the brain of your app.
* A controller handles user input, processes data, and returns a **view** or **data**.

✅ Example:

csharp

CopyEdit

public class HomeController : Controller

{

public ActionResult Index()

{

return View(); // goes to Views/Home/Index.cshtml

}

}

**📁 2. Models Folder**

* Contains **C# classes** that hold your **data structure** or **business logic**.
* Represents **database tables** or any data you're using.

✅ Example:

csharp

CopyEdit

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

}

**📁 3. Views Folder**

* Holds the **.cshtml** files (Razor files) — these generate the **HTML pages** the user sees.
* Organized by controller name.

✅ Example:

pgsql

CopyEdit

Views/

├── Home/

│ └── Index.cshtml

├── Shared/

│ └── \_Layout.cshtml <-- common layout for all pages

**📁 4. App\_Start Folder**

* Contains **configuration files** like:
  + RouteConfig.cs – handles **routing**
  + FilterConfig.cs – registers filters (like error handling)
  + BundleConfig.cs – bundles and minifies JS/CSS

**📁 5. Scripts Folder**

* Holds all **JavaScript** files including:
  + jquery.js
  + bootstrap.js
  + Your custom JS

Used for making pages interactive (dropdowns, validation, etc).

**📁 6. Content Folder**

* Contains **CSS stylesheets** and **images**
  + Example: Site.css, Bootstrap, background images, logos, etc.

**📁 7. App\_Data Folder**

* Used for **local database storage** (like .mdf files)
* Not always used, but useful for small DBs

**📄 8. Global.asax**

* Global application file
* Runs code when the app **starts** or **errors happen**
* Sets up routing and events

**⚙️ 9. Web.config**

* The **configuration file** for your whole web app
* Sets up:
  + Connection strings
  + Authentication
  + Session settings
  + Custom error pages

**🧠 MVC Flow Recap:**

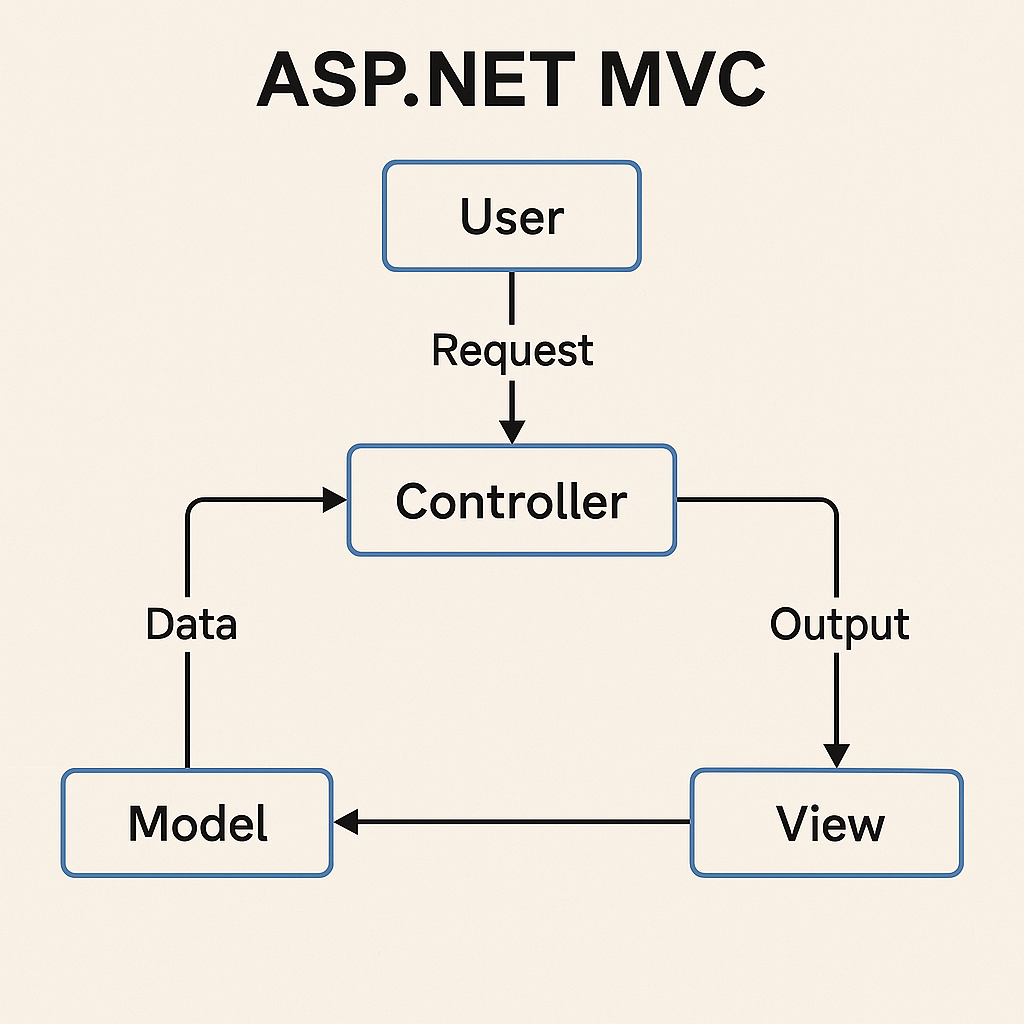
rust

CopyEdit

User --> Controller --> Model (Data) --> View (UI) --> Browser

**📝 Example Flow:**

User visits /Home/Index  
🔽  
**HomeController.cs** gets the request  
🔽  
Controller talks to the **Model** if needed  
🔽  
Controller sends data to the **View (Index.cshtml)**  
🔽  
View shows HTML to the user



**🧠 Basic Concepts**

**✅ What is a Controller?**

A **Controller** is the **brain** of your web application.

* It **handles user requests**
* It **talks to the model** (data)
* Then it **returns a view** (HTML page) to the user

📌 Example:

csharp

CopyEdit

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

**✅ What is an Action?**

An **Action** is just a **method inside the controller**.

* Each **action** handles one **task or page**
* Returns a view or data

📌 Example:

csharp

CopyEdit

public ActionResult Contact()

{

return View();

}

**✅ What is a View?**

A **View** is the **HTML page** that the user sees.

* It uses **Razor syntax** (@) to show data from the controller.

📌 Example: Views/Home/Contact.cshtml

html

CopyEdit

<h1>Welcome, @ViewBag.Name</h1>

**🔄 Value Passing from Action to View**

Here are 4 ways to pass data from a controller’s **action method** to a **view**:

**✅ 1. ViewBag**

📌 ViewBag is a **dynamic object** used to pass data from Controller to View.

**🔧 Example:**

csharp

CopyEdit

public ActionResult Index()

{

ViewBag.Message = "Hello from ViewBag!";

return View();

}

html

CopyEdit

<!-- Index.cshtml -->

<p>@ViewBag.Message</p>

**✅ Advantages:**

* Easy to use
* No need to define any class or model

**❌ Disadvantages:**

* **No IntelliSense** (autocomplete)
* Type issues may occur (it's dynamic)
* Works **only for one request**

**✅ 2. ViewData**

📌 ViewData is a **dictionary** object (key-value pair).

* You pass data like you do in a dictionary.

**🔧 Example:**

csharp

CopyEdit

public ActionResult Index()

{

ViewData["Message"] = "Hello from ViewData!";

return View();

}

html

CopyEdit

<!-- Index.cshtml -->

<p>@ViewData["Message"]</p>

**✅ Advantages:**

* Works in similar way as ViewBag
* Can store multiple values with keys

**❌ Disadvantages:**

* No IntelliSense
* Must cast to the correct type
* Works only for **current request**

**✅ 3. TempData**

📌 TempData is also a dictionary, but it **persists data for the next request**.

* Useful for **Redirects**

**🔧 Example:**

csharp

CopyEdit

public ActionResult First()

{

TempData["Notice"] = "You are redirected!";

return RedirectToAction("Second");

}

public ActionResult Second()

{

var msg = TempData["Notice"];

return View();

}

html

CopyEdit

<!-- Second.cshtml -->

<p>@TempData["Notice"]</p>

**✅ Advantages:**

* **Can persist across redirects**
* Great for **passing small messages (alerts, notices)**

**❌ Disadvantages:**

* Stored in session — not good for large data
* Can be lost if not accessed properly

**✅ 4. Model Binding (Strongly Typed View)**

📌 Pass a **model (class)** to the view — the cleanest and safest way.

**🔧 Model Class:**

csharp

CopyEdit

public class Student

{

public string Name { get; set; }

public int Age { get; set; }

}

**🔧 Controller:**

csharp

CopyEdit

public ActionResult Details()

{

Student s = new Student { Name = "Rifat", Age = 22 };

return View(s);

}

**🔧 View (Details.cshtml):**

csharp

CopyEdit

@model YourNamespace.Models.Student

<p>Name: @Model.Name</p>

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

**✅ Advantages:**

* **Type-safe**
* **IntelliSense** available
* Clean and maintainable

**❌ Disadvantages:**

* Need to create a model class
* A bit more setup than ViewBag/ViewData

**🔚 Summary Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Technique** | **Type-Safe?** | **Use Across Redirect?** | **Easy to Use?** | **IntelliSense** | **Best Use Case** |
| ViewBag | ❌ | ❌ | ✅ | ❌ | Small data for current view |
| ViewData | ❌ | ❌ | ✅ | ❌ | Similar to ViewBag, dictionary-style |
| TempData | ❌ | ✅ | ✅ | ❌ | Flash messages across actions |
| Model Binding | ✅ | ✅ (if used with TempData or Session) | ⚠️ (more setup) | ✅ | Passing structured data (best for real apps) |

**✅ Ways to Process Data from View to Action**

**🔹 1. HttpRequest Base Object**

This is the **rawest** way to get form data — using Request.Form["key"].

**✅ Example:**

**View (HTML):**

html

CopyEdit

<form method="post" action="/Home/Submit">

<input type="text" name="username" />

<button type="submit">Send</button>

</form>

**Controller:**

csharp

CopyEdit

[HttpPost]

public ActionResult Submit()

{

string user = Request.Form["username"];

return Content("Hello " + user);

}

**✅ Advantages:**

* Direct access to form data
* Works without needing models

**❌ Disadvantages:**

* Not type-safe
* Prone to errors (spelling, data type mismatch)

**🔹 2. FormCollection Object**

This is a slightly more structured version of Request.Form, using a FormCollection parameter.

**✅ Example:**

**View:**

html

CopyEdit

<form method="post" action="/Home/Submit">

<input type="text" name="username" />

<input type="text" name="email" />

<button type="submit">Send</button>

</form>

**Controller:**

csharp

CopyEdit

[HttpPost]

public ActionResult Submit(FormCollection form)

{

string name = form["username"];

string email = form["email"];

return Content($"Name: {name}, Email: {email}");

}

**✅ Advantages:**

* Slightly more organized than Request.Form
* Easy to loop through all form fields

**❌ Disadvantages:**

* Still not type-safe
* Still dependent on correct field names

**🔹 3. Variable Name Mapping (Parameter Matching)**

This is a **cleaner way** where the input names match method parameters directly.

**✅ Example:**

**View:**

html

CopyEdit

<form method="post" action="/Home/Submit">

<input type="text" name="username" />

<input type="number" name="age" />

<button type="submit">Submit</button>

</form>

**Controller:**

csharp

CopyEdit

[HttpPost]

public ActionResult Submit(string username, int age)

{

return Content($"User: {username}, Age: {age}");

}

Field names **must exactly match** the parameter names!

**✅ Advantages:**

* Type-safe
* Clean and readable
* No need to create a model

**❌ Disadvantages:**

* Limited to few fields
* Not ideal for complex data

**🔹 4. Model Binding with Class (Strongly Typed Model)**

The **best and most professional** method — bind all form fields into a model (class).

**✅ Step 1: Create a Model**

csharp

CopyEdit

public class User

{

public string Username { get; set; }

public int Age { get; set; }

}

**✅ Step 2: View**

html

CopyEdit

@model YourApp.Models.User

@using (Html.BeginForm("Submit", "Home", FormMethod.Post))

{

<input type="text" name="Username" />

<input type="number" name="Age" />

<button type="submit">Submit</button>

}

**✅ Step 3: Controller**

csharp

CopyEdit

[HttpPost]

public ActionResult Submit(User user)

{

return Content($"Hello {user.Username}, age {user.Age}");

}

**✅ Advantages:**

* **Type-safe**
* **Cleaner and scalable**
* Good for **forms with many fields**
* Easy validation with [Required], [EmailAddress], etc.

**❌ Disadvantages:**

* Requires model class
* More setup for small/quick forms

**🔚 Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Type-Safe** | **Clean Code** | **Best For** |
| Request.Form["key"] | ❌ | ❌ | Very basic form inputs |
| FormCollection | ❌ | ⚠️ | Simple multi-field forms |
| Variable Mapping | ✅ | ✅ | 1-3 input fields |
| Model Binding (class) | ✅✅ | ✅✅ | Complex forms (recommended) |

**✅ What is Annotation in ASP.NET MVC?**

**Annotations** are special tags (called attributes) that you write **above model properties** to add:

* Validation rules (like required, length, email format)
* Metadata (like display name)

They belong to the namespace:

csharp

CopyEdit

using System.ComponentModel.DataAnnotations;

**✅ Why use Data Annotations?**

To **automatically validate user input** without writing manual if-else checks.  
You write rules **once** in the model, and MVC uses them both:

* On the **server** (C#)
* On the **client** (JavaScript validation in browser)

**✅ Example with Explanation**

**🔷 Step 1: Create a Model with Annotations**

csharp

CopyEdit

using System.ComponentModel.DataAnnotations;

public class User

{

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

[StringLength(20, ErrorMessage = "Name can't be more than 20 characters")]

public string Name { get; set; }

[Range(18, 60, ErrorMessage = "Age must be between 18 and 60")]

public int Age { get; set; }

[EmailAddress(ErrorMessage = "Invalid Email format")]

public string Email { get; set; }

[DataType(DataType.Password)]

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

public string Password { get; set; }

[Compare("Password", ErrorMessage = "Passwords do not match")]

public string ConfirmPassword { get; set; }

}

**🔷 Step 2: Create the View (Razor Form)**

html

CopyEdit

@model YourApp.Models.User

@using (Html.BeginForm())

{

@Html.LabelFor(m => m.Name)

@Html.TextBoxFor(m => m.Name)

@Html.ValidationMessageFor(m => m.Name)

@Html.LabelFor(m => m.Age)

@Html.TextBoxFor(m => m.Age)

@Html.ValidationMessageFor(m => m.Age)

@Html.LabelFor(m => m.Email)

@Html.TextBoxFor(m => m.Email)

@Html.ValidationMessageFor(m => m.Email)

@Html.LabelFor(m => m.Password)

@Html.PasswordFor(m => m.Password)

@Html.ValidationMessageFor(m => m.Password)

@Html.LabelFor(m => m.ConfirmPassword)

@Html.PasswordFor(m => m.ConfirmPassword)

@Html.ValidationMessageFor(m => m.ConfirmPassword)

<button type="submit">Submit</button>

}

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

⚠️ @Scripts.Render("~/bundles/jqueryval") enables **client-side validation**. Make sure your project includes jQuery validation bundles.

**🔷 Step 3: Controller Action**

csharp

CopyEdit

[HttpPost]

public ActionResult Register(User user)

{

if (ModelState.IsValid)

{

// Save user, redirect, or show success

return Content("Registration successful!");

}

// If not valid, return the same view with errors

return View(user);

}

**✅ Common Data Annotations**

| **Annotation** | **Description** |
| --- | --- |
| [Required] | Field must not be empty |
| [StringLength(max)] | Max allowed characters |
| [Range(min, max)] | Value must fall in given range |
| [EmailAddress] | Must be a valid email format |
| [DataType(DataType.Password)] | Show as password input |
| [Compare("OtherField")] | Compares with another field (e.g., password) |
| [Display(Name = "Full Name")] | Custom label name |

**✅ Advantages**

* Clean code: validation logic stays in the model
* Less repeated code
* Automatic error display in view
* Both client-side and server-side validation

**❌ Disadvantages**

* Cannot handle **very complex conditions** (e.g., if A is true, then B must be X)
* Still need custom validation for some logic