**Introduction to ASP.NET**

**What is ASP.NET?**

ASP.NET is a **web development framework** developed by Microsoft for building **dynamic web applications, websites, and services**. It is an **open-source, server-side framework** that runs on **Windows, Linux, and macOS** using the **.NET Core** or **.NET Framework**.

**Key Features of ASP.NET**

1. **Cross-Platform Support**
   * Runs on **Windows, Linux, and macOS** (with ASP.NET Core).
2. **High Performance**
   * Optimized for **speed and scalability** using asynchronous programming.
3. **MVC Architecture**
   * Uses the **Model-View-Controller (MVC)** pattern for better code organization.
4. **Built-in Security**
   * Supports **authentication, authorization, and data protection**.
5. **Easy Integration**
   * Works with databases like **SQL Server, MySQL, and MongoDB**.
6. **Support for Web APIs**
   * Can be used to develop **RESTful APIs** for mobile apps and web services.

**ASP.NET Versions**

1. **ASP.NET Framework (Classic)**
   * Runs on **Windows**.
   * Supports **Web Forms, MVC, and Web API**.
2. **ASP.NET Core (Modern)**
   * Cross-platform (**Windows, Linux, macOS**).
   * **Lightweight, fast, and modular**.
   * Ideal for **cloud-based** and **modern web applications**.

**Components of ASP.NET**

1. **ASP.NET Web Forms**
   * Drag-and-drop UI design
   * Event-driven programming model
   * Good for **small applications**
2. **ASP.NET MVC (Model-View-Controller)**
   * Separation of concerns
   * Clean and testable architecture
   * Ideal for **large-scale applications**
3. **ASP.NET Web API**
   * Used to build **RESTful APIs**
   * Supports **JSON and XML data formats**
4. **ASP.NET Razor Pages**
   * A simple **page-based** approach
   * Uses **Razor syntax** (C# embedded in HTML)

**Conclusion**

ASP.NET is a **powerful** and **versatile** framework for web development, supporting modern programming practices and security features.

MVC is a design pattern used to build web applications. It separates an application into three main parts:

* **Model**: Handles data and business logic.
* **View**: Displays the user interface (UI).
* **Controller**: Manages user inputs and updates the Model and View.

Think of it like ordering food at a restaurant:

* **Model**: The kitchen where food (data) is prepared.
* **View**: The menu and the food you see on the table (UI).
* **Controller**: The waiter who takes your order and brings your food.

**How Does It Work?**

1. **User Requests Something**:
   * The user clicks a button or enters a URL (e.g., "Show me all products").
2. **Controller Handles Request**:
   * The Controller receives this request and decides what to do.
3. **Model Provides Data**:
   * The Controller asks the Model for the required data (e.g., a list of products).
   * The Model fetches the data from the database.
4. **View Displays Data**:
   * The Controller gives the data to the View.
   * The View displays the data as HTML.
5. **User Sees the Result**:
   * The user sees the list of products on the webpage.

Open source refers to software whose source code is made freely available for anyone to view, use, modify, and distribute. This approach promotes collaboration and transparency, allowing developers to contribute to the project and improve it over time. Open-source software is typically developed in a public, collaborative manner, often maintained by a community of developers.

**.NET Framework** and **.NET Core** are both development frameworks from Microsoft, but they serve different purposes and have some key differences:

| **Criteria** | **.NET Framework** | **.NET Core** |
| --- | --- | --- |
| **Platform Compatibility** | Windows only | Cross-platform (Windows, macOS, Linux) |
| **Application Types** | Desktop (Windows Forms, WPF), ASP.NET Web Apps | Web Apps, APIs, Microservices, Cloud Apps, Console |
| **Performance** | Slower due to legacy components | High performance and scalability |
| **Deployment** | Requires installation on the host machine | Self-contained deployment available |
| **Open Source** | Closed-source | Open-source and community-driven |
| **Development Model** | Monolithic | Modular with NuGet packages |
| **Versioning & Updates** | Tied to Windows updates | Independent and frequent updates |
| **Modern Development Support** | Limited (e.g., no containerization) | Full support (e.g., Docker, microservices) |
| **Successor** | No direct successor (legacy) | Continues as .NET 5, .NET 6, .NET 7, and beyond |

.NET Core achieves platform independence by:

* Compiling to Intermediate Language (IL) and using platform-specific JIT(just-in-time) compilation.
* Providing a consistent Base Class Library (BCL) across platforms.
* Allowing self-contained deployment with all necessary runtime components.
* Using abstraction layers for OS-specific functionality.

**Prefer or choose .NET Core if:**

* The project demands cross-platform integration.
* The project requires the development of microservices.
* The project relies heavily on CLI (Command Line Interface), as .NET Core is suitable for CLI.

**Prefer or choose .NET Framework if:**

* Applications are already running on .NET Framework.
* The applications require technologies like workflow, webforms, or WCF that are not present in .NET Core.
* Applications are built to run on Windows alone.

Directory Structure and Workflow of Asp.Net

**What is launchSettings.json in ASP.NET Core?**

* In the **Properties folder** in an ASP.NET Core project, you can find the **launchSettings.json** file, which contains **settings** that control how your web app is started on your development machine.
* The settings within this file are used when you run the .NET Core application either from **Visual Studio** or by using the **.NET Core CLI (Command Line Interface)**.
* The most important point to remember is that this **launchSettings.json** file is **only used within the local development machine.**
* This file is **not required** when publishing the ASP.NET Core application to the **production server.**

In **ASP.NET Core MVC**, the Program.cs file is the entry point of the application. It sets up the **web host** and **configures services** and **middleware** required for the application to run. Here's a breakdown of a typical Program.cs file:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

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

app.Run();

1. **WebApplication.CreateBuilder(args):**

* Initializes the application builder and sets up configuration, logging, and other essential services.

2. **AddControllersWithViews():**

* Registers MVC services, enabling the app to use controllers and views.

3. **app.Build():**

* Builds the WebApplication object.

4. **Middleware Configuration:**

* UseExceptionHandler() and UseHsts() are used for handling exceptions and enforcing HTTPS in non-development environments.
* UseHttpsRedirection() ensures that HTTP requests are redirected to HTTPS.
* UseStaticFiles() enables serving static files like CSS, JS, and images.
* UseRouting() sets up routing for the application.
* UseAuthorization() is used to enforce authorization rules.

5. **MapControllerRoute():**

* Configures the default route for MVC pattern, where the URL format is:

bash

/{controller}/{action}/{id?}

* + Defaults to HomeController and Index action if not specified.

6. **app.Run():**

* Starts the application and begins listening for incoming HTTP requests.

**Middleware** is software that's assembled(arranged in a sequence) into an application pipeline to handle requests and responses. It sits(is positioned in a specific place) between the incoming request and the outgoing response, processing data, handling authentication, logging, or even modifying the request/response.

* Ultimately, we need many pieces of middleware for an application to behave appropriately.
* Middleware has access to all the request and response.
* Order of middleware is very important.

In ASP.NET Core, both Use and Run are methods to configure middleware, but they behave differently in how they handle the request pipeline:

Use method : It **calls the next middleware** in the pipeline after executing its own logic.

Run Method : It **does not call the next middleware**. It acts as a terminal middleware.

Routing in ASP.NET Core is a mechanism that maps incoming requests to the appropriate controller actions. It determines how URLs are interpreted and how they connect to the application's endpoints.

* The Routing in ASP.NET Core MVC application is a mechanism in which it will inspect the incoming requests (i.e., URLs) and then map that request to the controllers and their action methods.
* This mapping is done by the routing rules which are defined for the application.
* We can do this by adding the Routing Middleware to the request processing pipeline.
* So, the ASP.NET Core Framework maps or connects the incoming requests (i.e., URLs) to the Controllers' action methods based on the routes configured in your application.

**Types of Routing in ASP.NET Core:**

1. **Conventional Routing**

**Conventional Routing** is a routing pattern in ASP.NET Core where routes are defined in a centralized location, typically in the Program.cs or Startup.cs file. It uses URL patterns to determine the controller and action method to handle an incoming request.

1. **Attribute Routing**

In Attribute-Based Routing, the [Route] attribute is used to define the routes.

The route is determined based on the attributes configured either at the controller level or at the action method level.

Both Conventional Based Routing and Attribute-Based Routing can be used in a single application.

Changing the controller or action name does **not** require the route template to be changed.

Token for the controller is [controller].

Token for the action method is [action].

Both tokens can be used together as [controller]/[action].

HTTP Methods in Routing (ASP.NET Core)

MapGet()

Maps HTTP GET requests.Used to retrieve data from the server.

app.MapGet("/items", () => "Get all items");

MapPost()

Maps HTTP POST requests.Used to create new resources on the server.

app.MapPost("/items", (Item item) => $"Item {item.Name} created");

MapPut()

Maps HTTP PUT requests. Used to update existing resources.

app.MapPut("/items/{id}", (int id, Item item) => $"Item {id} updated");

MapDelete()

Maps HTTP DELETE requests.Used to delete a resource from the server.

app.MapDelete("/items/{id}", (int id) => $"Item {id} deleted");

MapPatch()

Maps HTTP PATCH requests.Used for partial updates to a resource.

app.MapPatch("/items/{id}", (int id, Item item) => $"Item {id} patched");

**A View**

* A View provides the User Interface (UI) of the application to the user.
* A View is used to display content of an application and also to accept user inputs.
* View uses model data to create this UI.
* View contains both HTML markup and C# code that runs on the Web server.
* View has a file extension .cshtml.

**Razor**

* Razor is a syntax, based on the ASP.NET CORE Framework, that allows creating views.
* Razor is used to simplify the process of creating views.
* Razor is simple and easy to understand for users who are familiar with the C#.NET programming languages.

**Razor engine**

* The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.
* Razor engine is used as the default view engine by the MVC Framework.
* Compiles a view of your application when the view is requested for the first time.
* Delivers the compiled view for subsequent requests until you make changes to the view.
* Does not introduce a new set of programming language, but provides template markup syntax to segregate HTML markup and programming code in a view.
* First requires identifying the **server-side code** from the **markup code** to interpret the server-side code embedded inside a view file.
* Uses the @ symbol to separate the server-side code from the markup code.
* **While creating a Razor view, you should consider the following rules:**
  + Start inline expressions with @
  + Enclose code blocks between @{ and }
  + Variables are declared with the var keyword
  + Enclose strings in quotation marks
  + End a Razor code statement with a semicolon (;).
  + Use the .cshtml extension to store a Razor view file that uses **C#.NET** as the programming language.
  + Use the .vbhtml extension to store a Razor view file that uses **VB.NET** (Visual Basic .NET) as the programming language.

**Layout View (Master Page)**

* Provides a consistent look for all the views in a web application.
* Functions similarly to **ASP.NET Web Forms Master Pages**.
* Uses the .cshtml file extension.
* The default name of the Layout View is Layout.cshtml.
* Layout View files are typically located in the **Views/Shared** folder.
* An application can have multiple Layout Views.
* The Layout property is used to connect a Layout View with a specific view.

**Passing Data From Controller To View In ASP.NET Core 6**

* In an **ASP.NET CORE MVC** application, a controller typically performs the business logic of the application and needs to return the result to the user through a view
* You can use the following objects to pass data between the controller and the view:
  1. **ViewData**
  2. **ViewBag**
  3. **TempData**
  4. **Strongly Typed Views**

**ViewData**

It is used to pass data from controller to View.

Is a dictionary of objects that is derived from the **ViewDataDictionary** class.

**Some of the characteristics of ViewData are as follows:**

* The **life of a ViewData** object exists only during the **current request**.
* The value of ViewData becomes **null** if the request is **redirected**.
* ViewData requires **typecasting** when you use **complex data type** to avoid error.

The general syntax of **ViewData** is as follows:

**ViewData["<Key>"] = <Value>;**

where,

* **Key:** Is a String value to identify the object present in ViewData.
* **Value:** Is the object present in ViewData. This object may be a String, object, list, array, or a different type, such as **int, char, float, double, DateTime,** etc.

**Note:**

ViewData does not provide **compile time error checking**.

For example, if you **misspell keys**, you wouldn’t get any compile-time errors.

You get to know about the error **only at runtime**.

**ViewBag**

**The general syntax of ViewBag is as follows:**  
ViewBag.<PropertyName> = <Value>;

* **Where:**
  + **Property:** A string value representing a property of ViewBag.
  + **Value:** The assigned value, which can be a string, object, list, array, or another type (e.g., int, char, float, double, DateTime, etc.).

. **ViewBag** is a **dynamic data type property** of the base class of all controllers,

which is the ControllerBase class.

. ViewBag is a **dynamic data type** and internally uses ViewData to

store values.

. ViewBag exists **only for the current request** and becomes null if the

request is redirected.

. It is a **dynamic property** introduced in **C# 4.0**.

. ViewBag **does not require typecasting** when dealing with **complex data types**.

**Note:**

**.** ViewBag does not provide **compile-time error checking**.  
 For example, if you **misspell keys**, you wouldn’t get any compile-time errors.  
 You get to know about the error only at **runtime**.

**. ViewData** and **ViewBag** can access each other’s data interchangeably.

**TempData in ASP.NET MVC**

* TempData is used only for **current or subsequent requests** (short-lived).
* **Redirecting** is the primary use case (data persists after a request is killed and a new one is created).
* Facilitates sharing data between controller actions via the **TempData** dictionary.
* Derived from **TempDataDictionary** (stores data as **key-value pairs**).
* Values must be **type-cast** before use (e.g., string data = TempData["Key"] as string;).
* Always check for **null values** to prevent runtime errors.
* Data persists during **current request** and **immediate subsequent request** (e.g., after a redirect).
* Automatically discarded if not read in the next request.

Model

* A model is a class in C# (.cs file) that contains **properties** and **methods**.
* Example: A STUDENT model with properties like rollNo, Name, Gender, and Class.
* Models are used to **set or get data**.

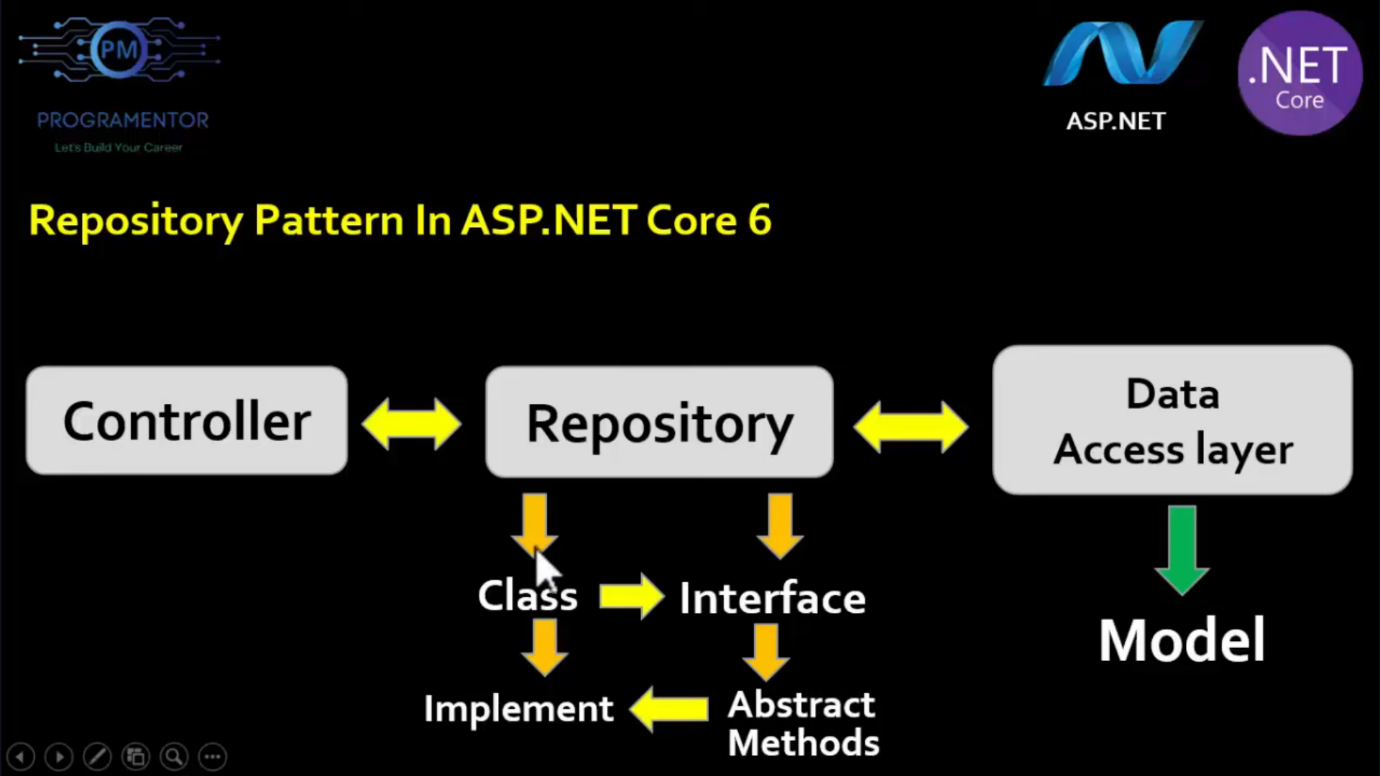
**When Do You Need a Model?**

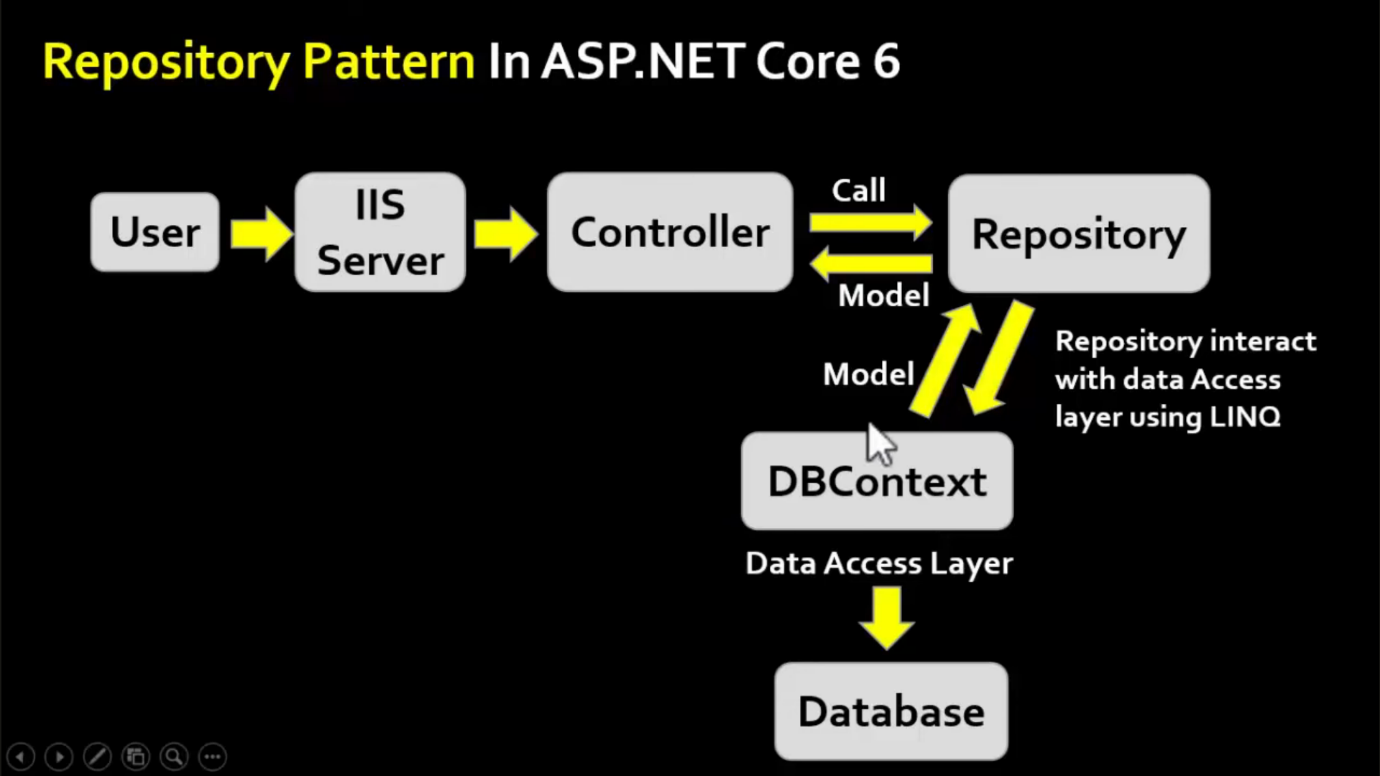
* If your application **does not** have data, you **don't need** a model.
* If your application **has** data, you **need** a model.
* Models consist of **a set of classes** that represent domain/business data.
* They also contain logic to **manage** this data.
* **Key takeaway:** Models are **used to manage the data** in an application.
* Data from models is **used in views**, while controllers manage the request flow.
* Models allow performing **CRUD operations**.
* Although not mandatory, it is a **good practice** to store all model classes in the **Models folder**.

The **Repository Pattern** is a design pattern that helps manage data access in a structured way by **separating the data logic from business logic**. It provides an abstraction over data access, making the application **more maintainable, testable, and scalable**.

The Repository Pattern is like having a smart helper that manages all your data for you. It keeps your code clean, makes it easier to change how you store data, and helps you reuse code. In ASP.NET MVC, it's a great way to keep your website organized and easy to maintain.

1. **Separation of Concerns** – Keeps database logic separate from business logic.
2. **Improved Maintainability** – Changes in the data source (e.g., switching from SQL Server to MongoDB) don't affect business logic.
3. **Testability** – Makes unit testing easier by using dependency injection and mocking data access.
4. **Encapsulation of Queries** – Centralizes database queries inside repositories, preventing duplicate code.





**Strongly Typed View (Text from Images)**

* A **strongly typed view** or **strongly typed object** is used to pass data from the controller to a view.
* The **view which binds with any model** is called a strongly typed view.
* You can bind any **class as a model to a view**.
* You can access **model properties** on that view.
* You can use **data associated with the model** to render controls.
* A view that is designed by targeting a **specific model class object** is called a **"Strongly Typed View."**
* In a strongly typed view, the **view is bound** to a corresponding **model class object** or a **list of objects**.

the \_ViewImports.cshtml file is used to **define common directives** that can be shared across multiple Razor views. This helps in avoiding repetitive code in each view.

Tag Helper

**Tag Helpers** in ASP.NET Core MVC provide a **way to create reusable HTML elements** using **server-side logic**. They allow developers to work with **HTML-friendly syntax** instead of writing raw C# code inside Razor views.

* **Tag helpers** are basically **special attributes** provided by ASP.NET Core.
* Tag Helpers enable server-side components to participate in creating and rendering HTML elements in **Views**.
* **Tag helpers** are a new feature and similar to **HTML helpers (MVC 5)**, which help us render HTML.
* There are many **built-in Tag Helpers** for common tasks, such as **creating forms, hyperlinks, loading assets**, etc.
* Tag Helpers are authored in **C#**, and they target HTML elements based on the element name.
* For example, the built-in **LabelTagHelper** can target the HTML <label> element when the **LabelTagHelper** attributes are applied.
* Before starting work with tag helpers, make sure you have included the **namespace** for tag helpers in your **ViewImports** file.
* The required namespace is:  
  **Microsoft.AspNetCore.Mvc.TagHelpers**.
* Add this line in your **ViewImports** file:

Entity Framework Core

**Entity Framework Core** is the redesigned version of Entity Framework after **EF 6**. It is:

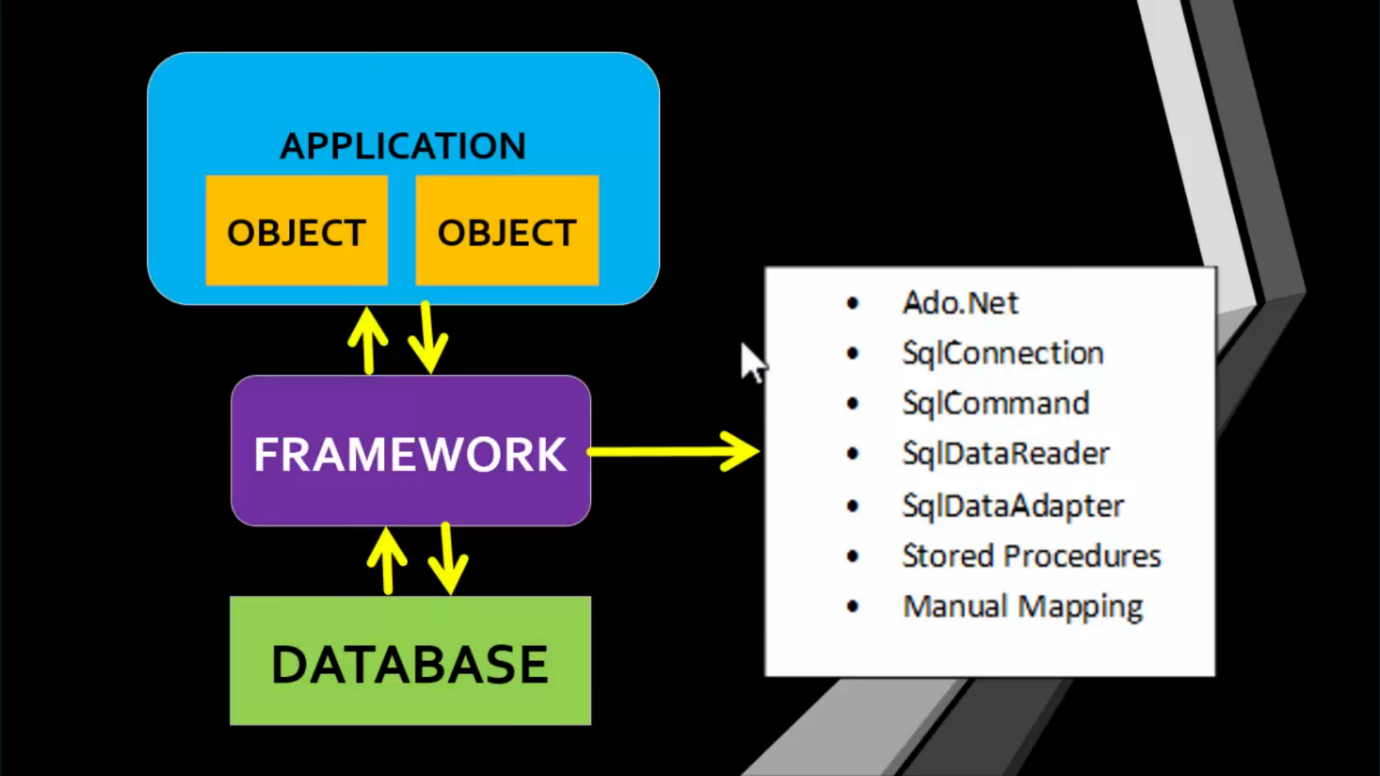
* **Open-source**, lightweight, extensible, and cross-platform.
* A modern data access technology for **.NET**.
* Entity Framework is an **Object/Relational Mapping (ORM) framework**.
* It is an **enhancement to ADO.NET** that gives developers an automated mechanism for **accessing & storing** the data in the database.
* **EF Core** is intended to be used with **.NET Core applications**. However, it can also be used with standard **.NET 4.5+ framework**-based applications.

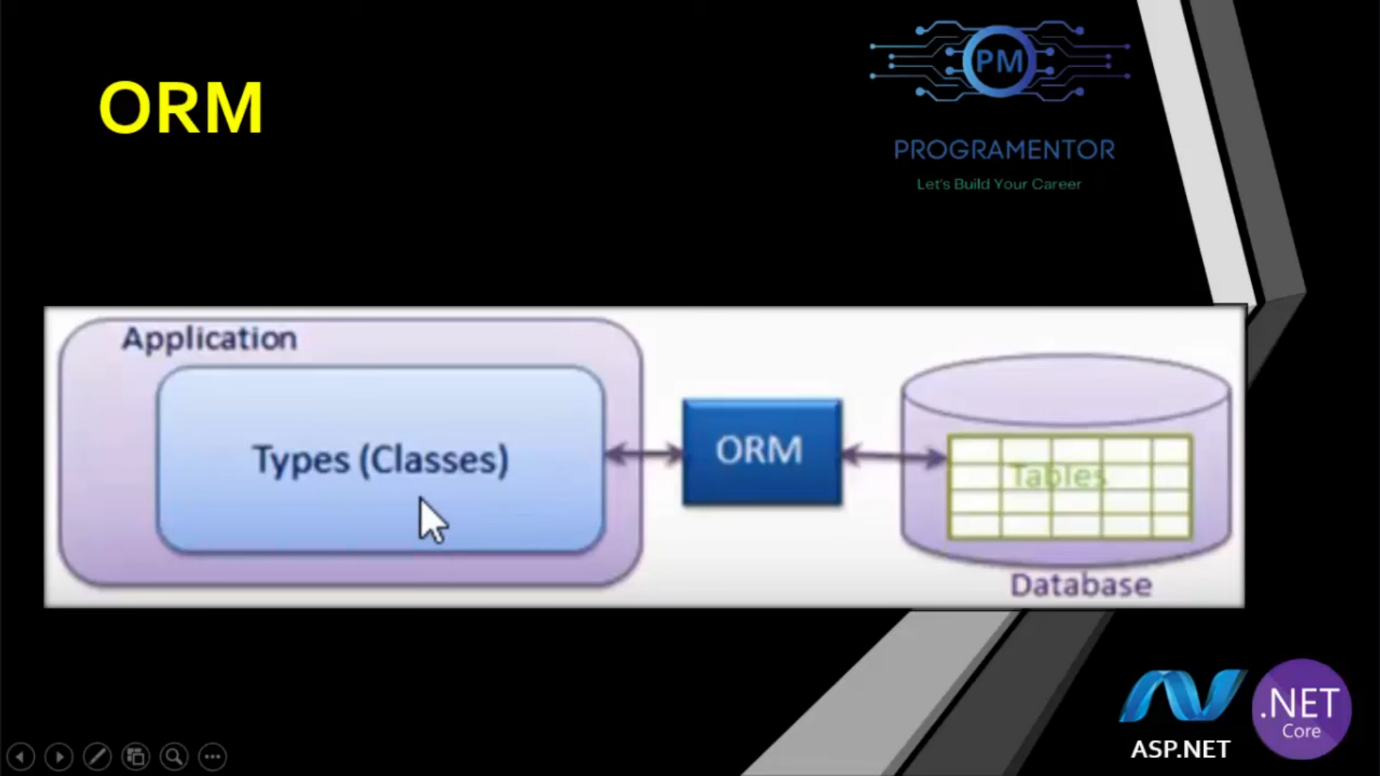
An ORM simplifies the process of accessing data from applications. It is a tool for mapping data from domain objects (model classes) to a relational database (e.g., **SQL Server DB**).

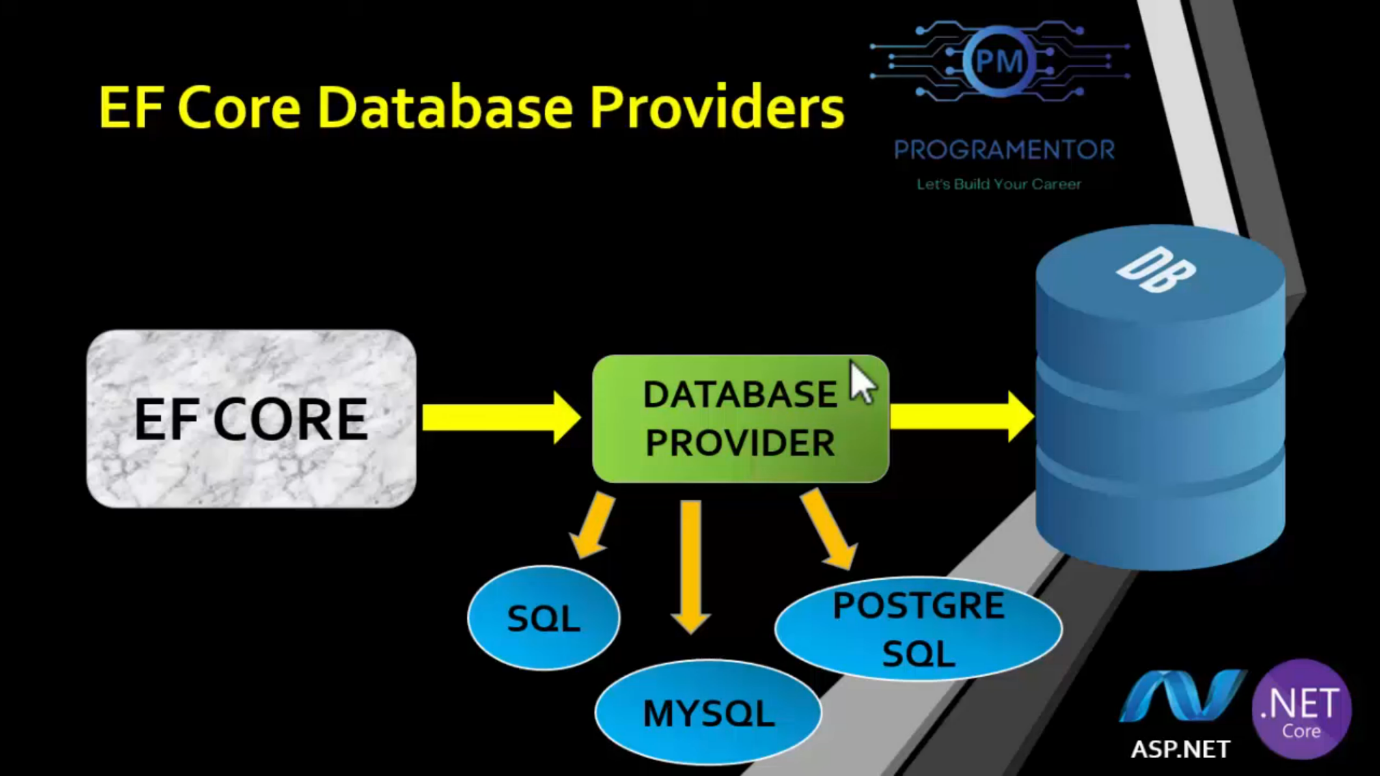
EF Core is an **Object/Relational Mapping (ORM)** framework that enhances **ADO.NET** by providing developers an automated mechanism for accessing and storing data in databases.

**Key Takeaways**

* ORM bridges domain models (e.g., C# classes) and relational databases.
* EF Core streamlines data operations, reducing manual SQL coding.
* Supports cross-platform development and modern .NET ecosystems.





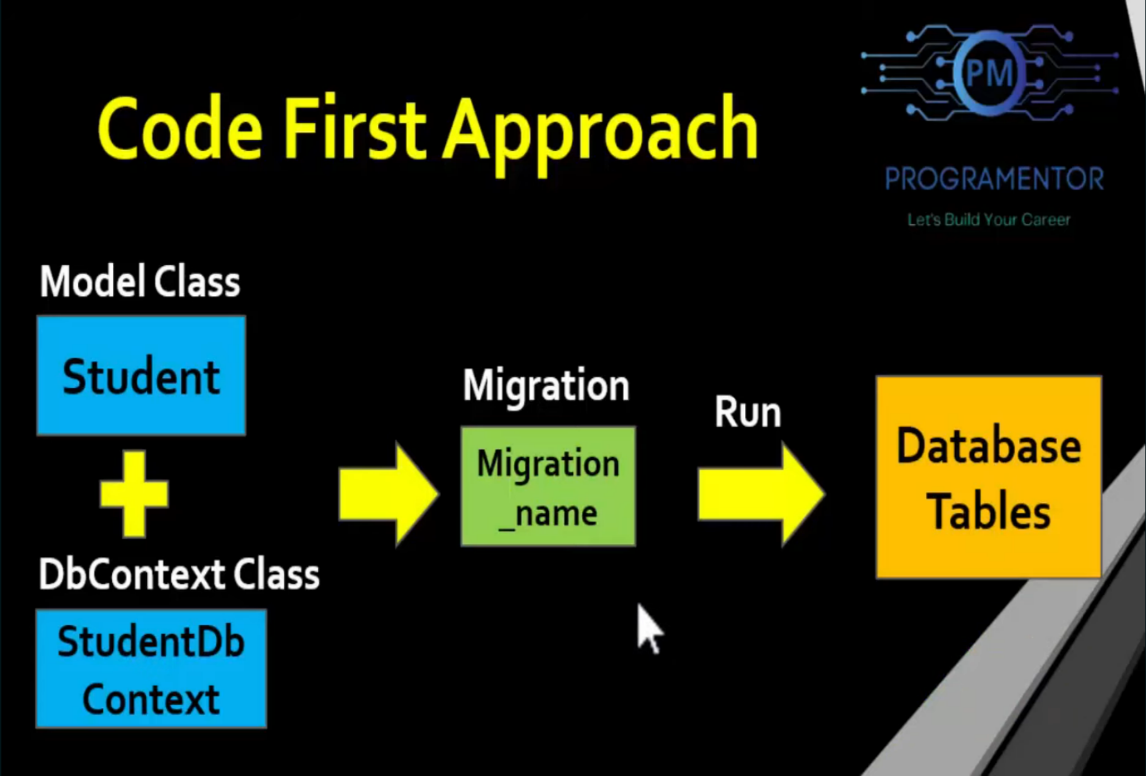


**EF Core Development Approaches**

* **EF Core supports two development approaches:**
  1. **Code-First**

The code-first approach allows you to:

* + 1. Define your own model by creating custom classes.
    2. Generate a database based on these models.

****

The **Code-First Approach** in Entity Framework Core allows developers to create and manage databases using C# classes instead of designing the database manually. Each step in this process serves a specific purpose in setting up and managing a database.

**Step 1: Install Required Packages**

Before using Entity Framework Core, we need to install necessary NuGet packages that provide the required functionalities.

Install-Package Microsoft.EntityFrameworkCore

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

* Microsoft.EntityFrameworkCore → Provides core EF functionalities.
* Microsoft.EntityFrameworkCore.SqlServer → Enables SQL Server as the database provider.
* Microsoft.EntityFrameworkCore.Tools → Enables commands like Add-Migration and Update-Database.

**Step 2: Create Model and DbContext Classes**

This step involves defining the **Model** (which represents a table) and **DbContext** (which manages the database).

**Model Class (Represents a Table)**

public class Product

{

public int Id { get; set; } // Primary Key

public string Name { get; set; }

public decimal Price { get; set; }

}

* Each property in this class corresponds to a column in the database table.
* Id is automatically recognized as the **Primary Key** by EF Core.

**DbContext Class (Manages Database)**

using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options)

{

}

}

* ApplicationDbContext manages **database connections and queries**.
* DbSet<Product> Products represents a **table** in the database.
* This class acts as a **bridge between our application and the database**.

**Step 3: Add Connection String in appsettings.json**

A **Connection String** tells EF Core how to connect to the database.

{

"ConnectionStrings": {

"DefaultConnection": "Server=YOUR\_SERVER\_NAME;Database=ProductDB;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

* "Server=YOUR\_SERVER\_NAME" → Specifies the **SQL Server name**.
* "Database=ProductDB" → Specifies the **database name**.
* "Trusted\_Connection=True" → Uses **Windows Authentication** to connect.

**Step 4: Register DbContext in Program.cs**

To use ApplicationDbContext, we need to **register it in the application’s services**.

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Register DbContext

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();

app.Run();

**How It Works?**

* builder.Services.AddDbContext<ApplicationDbContext> → Registers the database context.
* options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")) → Connects to **SQL Server** using the **connection string**.
* This ensures that our application can communicate with the database.

**Step 5: Add and Apply Migrations**

Once the models and DbContext are ready, we need to **create and apply database migrations**.

**What is a Migration?**

A **migration** is a snapshot of our database schema. EF Core **converts C# classes into database tables** using migrations.

**Command 1: Add Migration**

Add-Migration InitialCreate

* Creates a migration file that contains SQL commands for creating tables.
* This migration file is stored in the Migrations folder.

**Command 2: Update Database**

Update-Database

* Executes the SQL commands in the migration file.
* **Creates the database and tables** based on our models.

Now, the database is ready for use! 🎉

**Step 6: Perform CRUD Operations**

Once the database is set up, we can use the ApplicationDbContext class to perform CRUD (Create, Read, Update, Delete) operations.

**Create (Insert Data)**

using (var context = new ApplicationDbContext())

{

var product = new Product { Name = "Laptop", Price = 1200 };

context.Products.Add(product);

context.SaveChanges();

}

* Creates a new Product object.
* Adds it to the Products table.
* Calls SaveChanges() to insert the data into the database.

**Read (Retrieve Data)**

using (var context = new ApplicationDbContext())

{

var products = context.Products.ToList();

foreach (var product in products)

{

Console.WriteLine($"{product.Id} - {product.Name} - {product.Price}");

}

}

* Fetches all records from the Products table.
* Loops through the results and prints them.

**Update (Modify Data)**

using (var context = new ApplicationDbContext())

{

var product = context.Products.FirstOrDefault(p => p.Id == 1);

if (product != null)

{

product.Price = 1500;

context.SaveChanges();

}

}

* Finds the product with Id = 1.
* Updates its Price.
* Calls SaveChanges() to apply changes.

**Delete (Remove Data)**

using (var context = new ApplicationDbContext())

{

var product = context.Products.FirstOrDefault(p => p.Id == 1);

if (product != null)

{

context.Products.Remove(product);

context.SaveChanges();

}

}

* Finds the product with Id = 1.
* Removes it from the database.

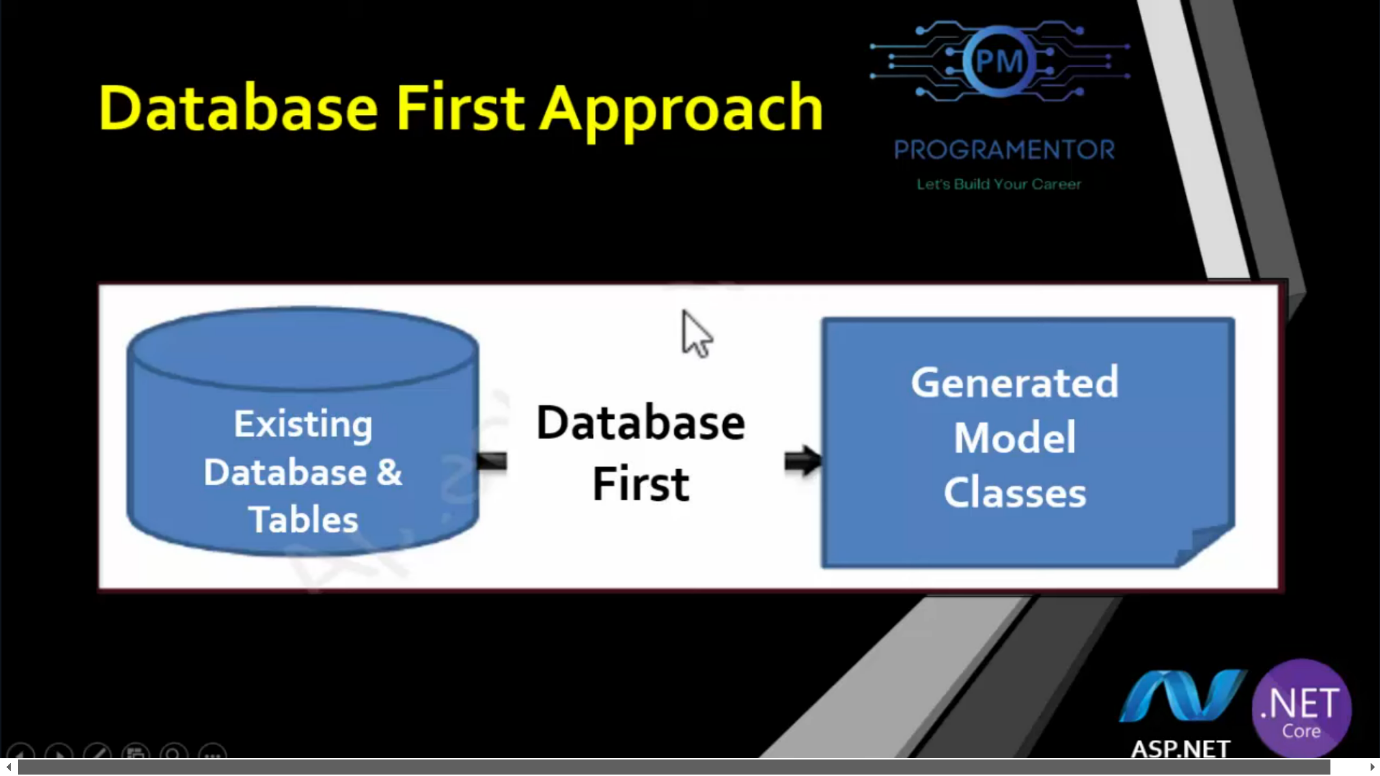
**Final Summary**

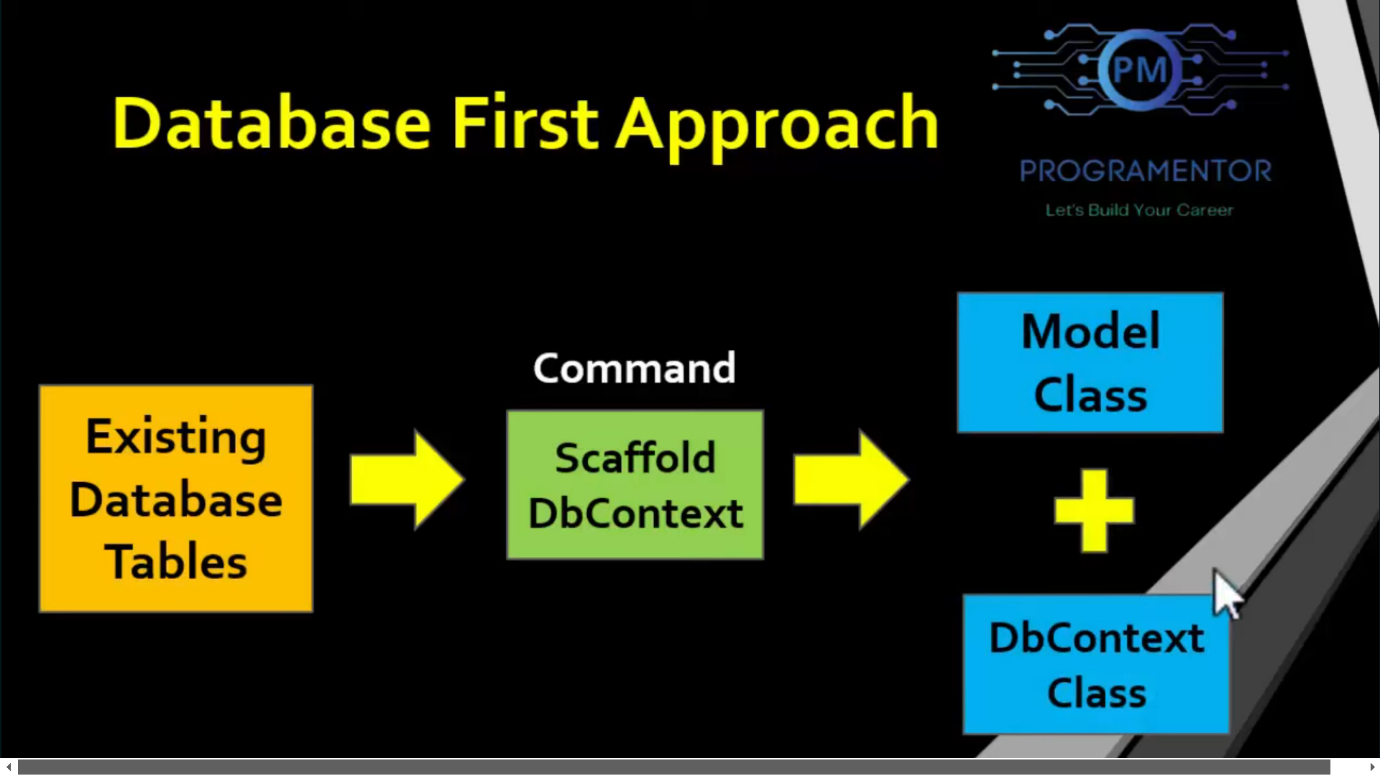
| **Step** | **Description** |
| --- | --- |
| **Step 1** | Install EF Core packages using NuGet. |
| **Step 2** | Create **Model classes** (representing database tables) and **DbContext class** (managing database). |
| **Step 3** | Configure **Connection String** in appsettings.json. |
| **Step 4** | Register DbContext in Program.cs. |
| **Step 5** | Run Add-Migration and Update-Database to create the database. |
| **Step 6** | Perform **CRUD operations** (Create, Read, Update, Delete). |

**Why Use the Code-First Approach?**

✅ **Flexibility** → Define models using C# instead of designing the database manually.  
✅ **Version Control** → Migrations track database changes, making it easy to update.  
✅ **Automated Schema Updates** → Any change in the model updates the database without manual SQL queries.

**Database First Approach**  
The database-first approach involves Entity Framework Core generating model classes and properties based on an existing database structure (tables, columns, etc.). This approach is suitable when the database is already in place.





**Steps to Implement Database-First Approach**

**1st Step: Install Required Packages**  
Install the following NuGet packages in your ASP.NET Core MVC application:

1. Microsoft.EntityFrameworkCore.SqlServer
2. Microsoft.EntityFrameworkCore.Tools
3. Microsoft.EntityFrameworkCore.Design

**2nd Step: Scaffold DbContext and Models**  
Run the following command in the Package Manager Console to generate model classes and the DbContext class:

Scaffold-DbContext "Server=LAPTOP-BV1V3PVK\SQLEXPRESS;Database=CodeFirst;Integrated Security=True;TrustServerCertificate=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

**Updating Models After Database Changes**  
If the database schema is updated, re-run the scaffold command with the -force flag to overwrite existing files:

Scaffold-DbContext "server=ServerName; database=DatabaseName; trusted\_connection=true" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Force

**3rd Step: Move Connection String to appsettings.json**  
Add your connection string to appsettings.json:

{

"ConnectionStrings": {

"DefaultConnection": "server=ServerName; database=DatabaseName; trusted\_connection=true"

}

}

**4th Step: Register Connection String in Program.cs**  
Update Program.cs to read the connection string from appsettings.json and configure the DbContext:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<YourDbContext>(options =>

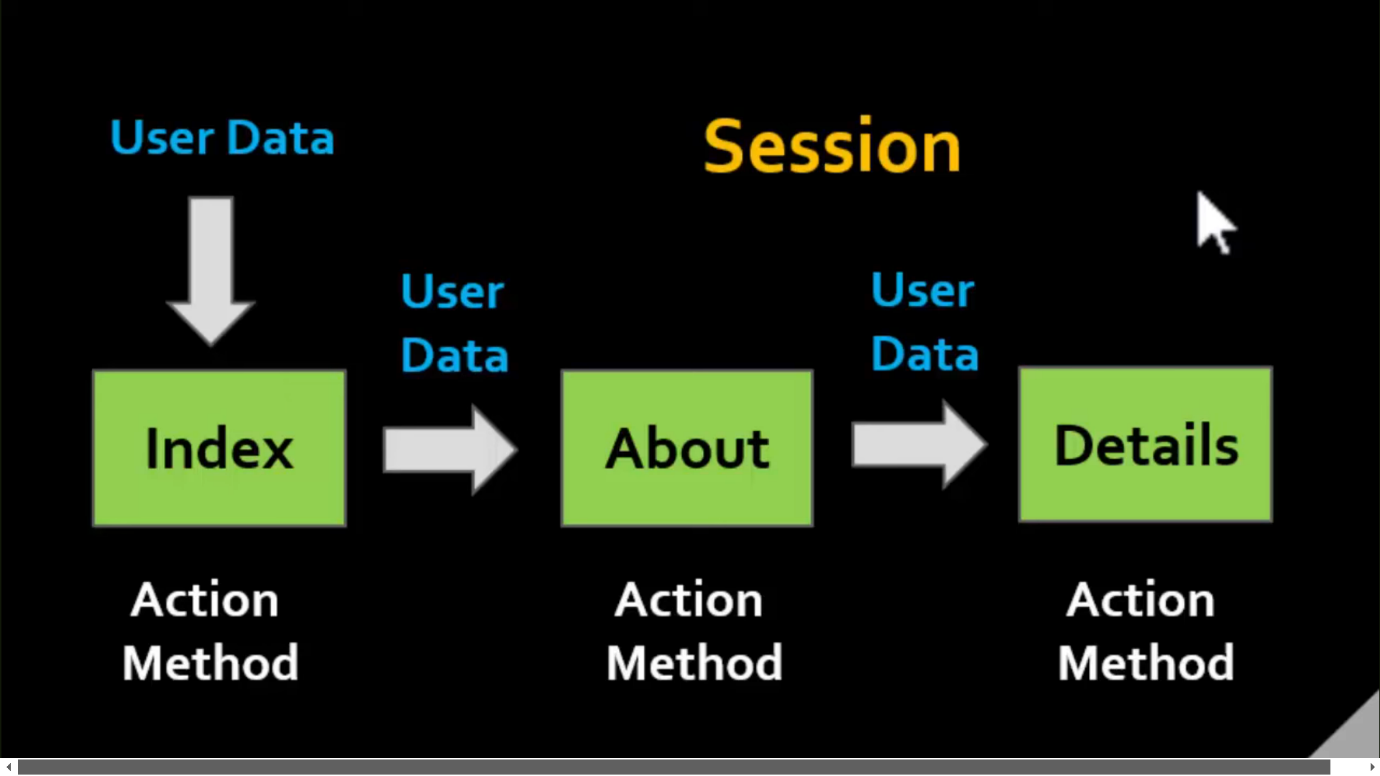
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

Replace YourDbContext with the name of your generated DbContext class.

A **session in ASP.NET** is a mechanism for storing user-specific data across multiple requests. It allows web applications to maintain state for individual users. Session data is stored on the server and can be accessed using the Session object.

**Key Points:**

* **Storage:** Can be in memory (InProc), a state server (StateServer), a database (SQL Server), or a custom provider.
* **Usage:**
  + Store: Session["key"] = value;
  + Retrieve: var data = Session["key"];
  + Remove: Session.Remove("key");
* **Timeout:** Default is 20 minutes, configurable via Session.Timeout.
* Session is a server-side state management technique for storing user-specific data while they browse a web app.
* Data is persisted across requests using a cache (ephemeral storage).
* Sessions are browser-specific: Each browser has a unique session, and sessions are not shared across browsers.
* Sessions expire when the browser session ends or after a set time (default: 20 minutes).
* Critical data should be stored in a database; sessions act as a performance optimization cache.
* Data is stored in key-value pairs and is valid for all requests (not just a single redirect).
* Each session has a unique session ID.



**Steps to Configure Session in ASP.NET Core 6**

**Step 1: Add Session Middleware**

* Add the following line **before building** the app:

builder.Services.AddSession();

* Add the following line **after building** the app:

app.UseSession();

**Step 2: Create a Session Variable**

HttpContext.Session.SetString("MyKey", "MyValue");

// Other data types (e.g., integers, objects) can also be stored.

**Step 3: Access a Session Variable**

var value = HttpContext.Session.GetString("MyKey");

**What is an API?**

An API is like a **messenger** or **waiter** that lets two apps or systems talk to each other. You don’t need to know *how* the other system works—you just ask for what you want, and the API delivers it!

**Real-World Examples**

**1. Weather Apps**

* **How it works:** Apps like *Weather.com* or your phone’s weather app don’t create weather data themselves. They use APIs to fetch data from big weather stations.
* **Example:** You type "Mumbai weather," and the app uses an API to ask a weather service for data, then shows it to you.

**2. Instagram Stories with Music**

* **How it works:** When you add a song to your Instagram Story, Instagram uses an API to connect to Spotify or Apple Music. You pick the song, and the API "delivers" it to your Story.

**3. Online Food Delivery (Swiggy/Zomato)**

* **How it works:** When you order food, the app uses APIs to:
  + Show restaurant menus (from the restaurant’s system).
  + Track your delivery (using GPS APIs).
  + Process payments (via PhonePe/Paytm APIs).

**What is a Web API?**

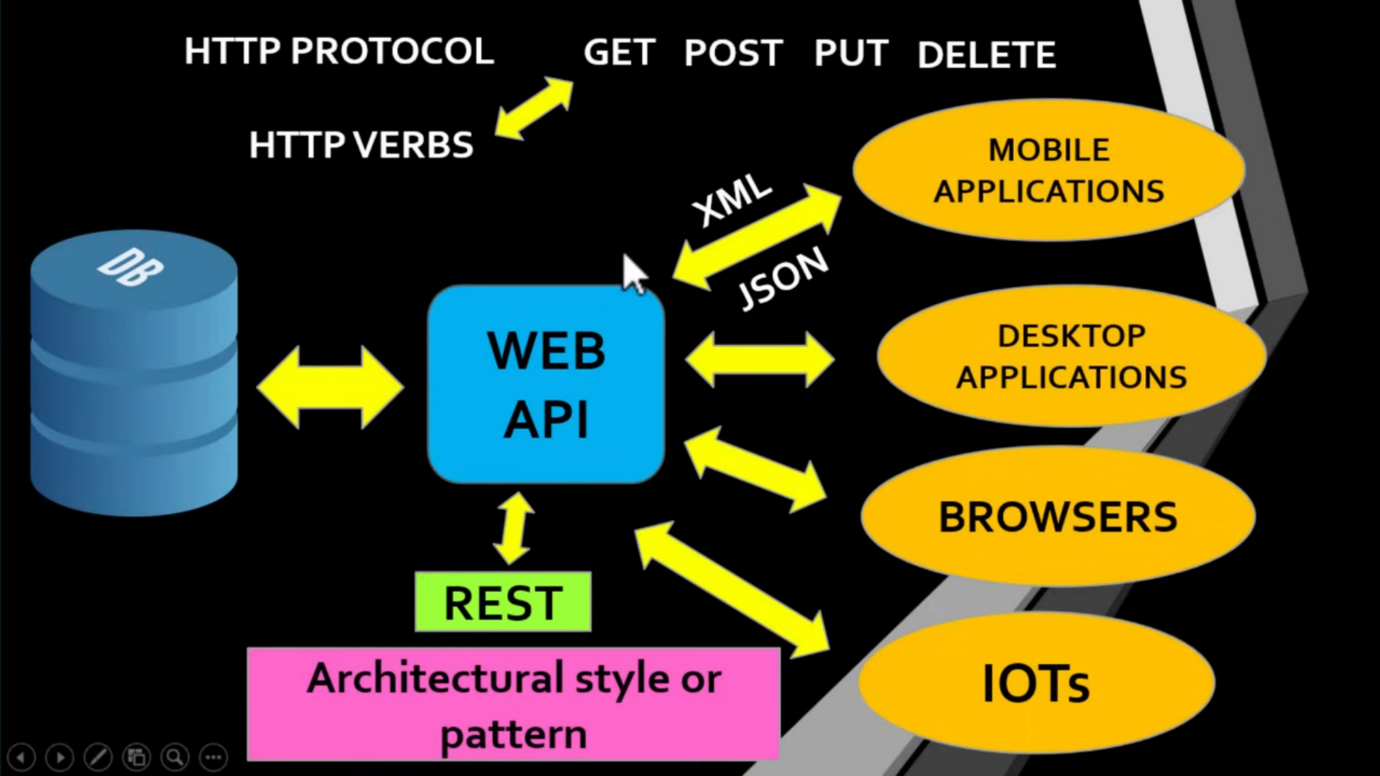
A **Web API** is a **specific type of API** that uses the **internet (HTTP/HTTPS)** to communicate. It’s like a waiter who works **only for online orders** (via websites or apps).

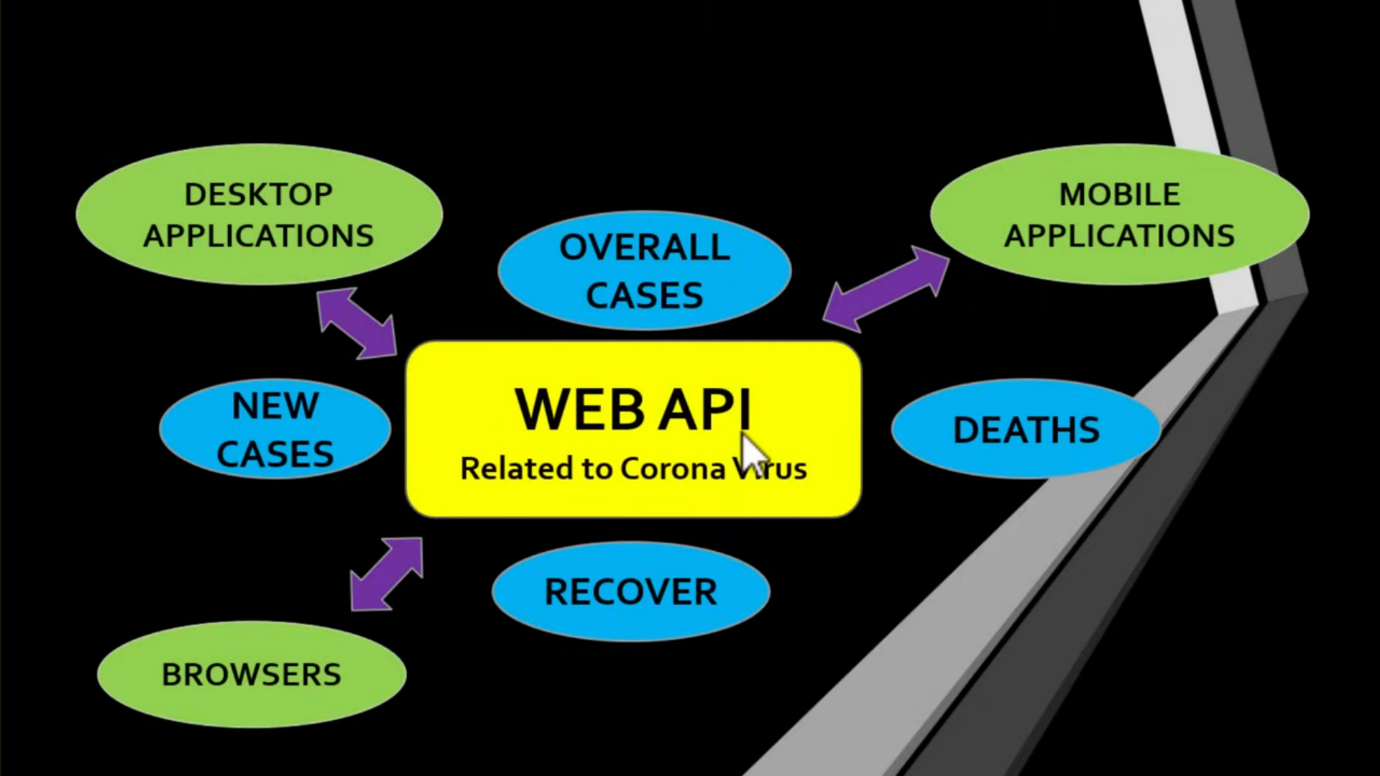
**Key Features of Web APIs:**

1. Works over the **web** (uses URLs like https://api.weather.com/data).
2. Sends/receives data in formats like **JSON** or **XML**.
3. Used by apps, websites, or services that need to talk to servers far away (e.g., cloud servers).

**How is a Web API Different from a Regular API?**

| **Feature** | **Regular API** | **Web API** |
| --- | --- | --- |
| **Communication** | Can work offline (e.g., within your device). | Always uses the **internet** (HTTP/HTTPS). |
| **Example** | Your phone’s calculator app using math functions from its own code. | Instagram fetching posts from Facebook’s servers. |
| **Access** | Limited to one system or local network. | Accessible globally via URLs. |
| **Data Format** | Can use any format (not always JSON/XML). | Usually uses **JSON** or **XML**. |





**RESTful Architecture**

* Follows the principles of **Representational State Transfer (REST)**.
* Uses HTTP methods like GET, POST, PUT, and DELETE to interact with resources identified by URLs (Uniform Resource Locators).
* A resource is any information with a name (e.g., image, entity, document).
* Resources can be represented in formats like **Text**, **JSON**, or **XML** (JSON is the most popular).

**RESTful System Components**

1. **Client**: Requests resources.
2. **Server**: Holds and manages resources.

**REST Constraints**

1. Uniform Interface
2. Client-Server Separation
3. Stateless
4. Cacheable
5. Layered System
6. Code on Demand

**What is Swagger?**

* Swagger is used to describe **RESTful APIs**.
* Allows developers to create **interactive and human-readable API documentation**.
* Used to **test APIs**.

**What is ControllerBase?**

* The ControllerBase class is a **base class for controllers** in **ASP.NET Core** that handles **HTTP requests**.
* Provides common properties and methods for controllers to handle HTTP requests and generate HTTP responses.

**[ApiController] Attribute**

* The [ApiController] attribute enables features in ASP.NET Core, including:
  1. **Attribute routing requirement** (routes defined via attributes).
  2. **Automatic model validation** (checks if incoming data is valid).
  3. **Binding source parameter inference** (automatically determines where parameters come from, e.g., query string or body).

Identity Framework

The **ASP.NET Core Identity Framework** is a robust, extensible system for managing user authentication, authorization, and identity-related features in ASP.NET Core applications. It provides out-of-the-box solutions for user management (registration, login, roles, claims, passwords, etc.) and integrates seamlessly with ASP.NET Core's security middleware.

**1️. Password-Based Authentication (Default)**

Users log in with a **username/email and password** stored in the Identity database.

**Pros:**  
1. Simple and easy to implement.

2. Identity Framework automatically hashes passwords for security.

3. Supports **password reset** and **account lockout**.

**Cons:**  
1. Users must remember passwords.  
2. If passwords are weak, accounts can be hacked.  
3. Managing password policies can be complex.

**2️. Two-Factor Authentication (2FA)**

An extra security layer where users need a **password + a second factor** (OTP, authenticator app, SMS, email).

**Pros:**  
1. More secure than just using passwords.  
2. Reduces the risk of **stolen passwords** being misused.  
3. Identity Framework has built-in support.

**Cons:**  
1. Extra step for users (slightly inconvenient).  
2. If the second factor (SMS, Email) is compromised, it can be bypassed.

**3️ Social Authentication (OAuth & OpenID Connect)**

Users log in using **Google, Facebook, Twitter, Microsoft, GitHub, etc.**

**Pros:**  
1. Users don’t need to remember new passwords.  
2. Secure, as these providers handle authentication.  
3. Faster login experience.

**Cons:**  
1. Requires setting up OAuth credentials (Google, Facebook, etc.).  
2. If the provider (Google, Facebook) goes down, users can't log in.

**4️ Token-Based Authentication (JWT)**

Used for **API authentication** where users get a **JWT (JSON Web Token)** instead of a session cookie.

**Pros:**  
1. Perfect for mobile apps & SPAs (React, Angular, etc.).  
2. Stateless (no session needed on the server).

3. Secure (tokens expire and can’t be reused easily).

**Cons:**  
1. If a token is leaked, the account is compromised.  
2. Tokens need to be refreshed manually (or use refresh tokens).

**5️ Windows Authentication**

Uses **Active Directory (AD)** credentials, mainly for intranet applications in organizations.

**Pros:**  
1. No need for users to enter credentials (if logged into Windows).  
2. Secure (uses domain authentication).  
3. Best for enterprise applications (company networks).

**Cons:**  
1. Only works for **Windows-based** networks.  
2. Not useful for public websites.

**6️ API Key Authentication**

Used when an **API key** (instead of a username/password) is sent in API requests.

**Pros:**  
1. Simple to implement for APIs.  
2. No need for user sessions.

**Cons:**  
1. If an API key is leaked, the API can be misused.  
2. No built-in expiration unless manually handled.

**7️ External Identity Providers (Azure AD, Okta, Auth0)**

Used for **enterprise authentication** where organizations use **Azure Active Directory (Azure AD)**, **Okta**, or **Auth0**.

**Pros:**  
1. Best for large companies needing single sign-on (SSO).  
2. Very secure with enterprise-level policies.

**Cons:**  
1. Requires a **third-party** identity provider (paid service).  
2. Setup can be complex.

**Azure AD** → Best for **enterprise authentication & Microsoft environments**.

**Okta** → Best for **secure, scalable authentication with multiple integrations**.

**Auth0** → Best for **developer-friendly authentication with social logins**.

**Google Workspace** → Best for **Google-based authentication**.

**Amazon Cognito** → Best for **AWS-powered applications needing authentication**.

**Certificate-Based Authentication (CBA)?**

**Certificate-Based Authentication (CBA)** is a secure authentication method where users verify their identity using a **digital certificate** (like **X.509 Certificate**) instead of a username and password. The **certificate** contains user information and is issued by a **trusted Certificate Authority (CA)**. The server validates the certificate, and if it's **valid**, access is granted; otherwise, it's denied.

8. Certificate-Based Authentication (CBA), users authenticate themselves using a digital certificate (X.509) instead of a password.  
Example: Smart Cards, Digital Signature, SSL Certificate.

Pros:

1. Highly secure (No password required).
2. Prevents phishing and hacking.
3. Can be used for device authentication as well.

**Cons:**

1. Complex setup and certificate management.
2. If a certificate is stolen, it can be misused.
3. Expensive to maintain.

| **Authentication Type** | **Pros** | **Cons** |
| --- | --- | --- |
| **Password-Based** | Simple and widely used | Weak security, password leaks |
| **Token-Based (JWT)** | Stateless and fast | Token theft, expiry issues |
| **Multi-Factor (MFA)** | Highly secure | Time-consuming, complex setup |
| **Certificate-Based** | No password needed | Complex infrastructure |
| **OAuth2.0** | Quick social login | Third-party dependency |
| **OIDC (OpenID)** | Both auth and authorization | Requires identity provider |
| **Windows Auth** | Seamless login in domain | Not for public use |
| **SAML** | Single Sign-On (SSO) | Complex setup |
| **Social Media Login** | Fast login | Dependency on social platforms |
| **Biometric** | Highly secure | Requires hardware |

**Stateless** means **the server does not store client session data** between requests. Each request **must contain all necessary information** for the server to process it.

**OIDC** → The **protocol** that defines how authentication works.

**OpenID Connect (OIDC)** is an **authentication protocol** that works on top of **OAuth 2.0**. It is used to:

1. Authenticate users (verify their identity)  
2. Provide a **secure way** to get user information (like email, name, etc.)  
3. Issue **ID Tokens** (which are JWT tokens containing user data)

**Auth0** → A **service** that makes implementing OIDC much easier.

| **Scenario** | **Best Authentication Method** |
| --- | --- |
| Public Website (E-commerce, Blog) | **OAuth2.0 (Google, Facebook)** |
| Company Intranet | **Windows Authentication (Active Directory)** |
| Highly Secure App (Banking, Healthcare) | **Certificate-Based + MFA** |
| Large Organization (SSO) | **SAML or OpenID Connect** |
| IoT Devices | **Certificate-Based Authentication** |
| Mobile App API | **JWT Token-Based Authentication** |