ASP.NET Core

Presentation by Rohan Gope

Table of Contents

ASP.NET Core Prerequisites

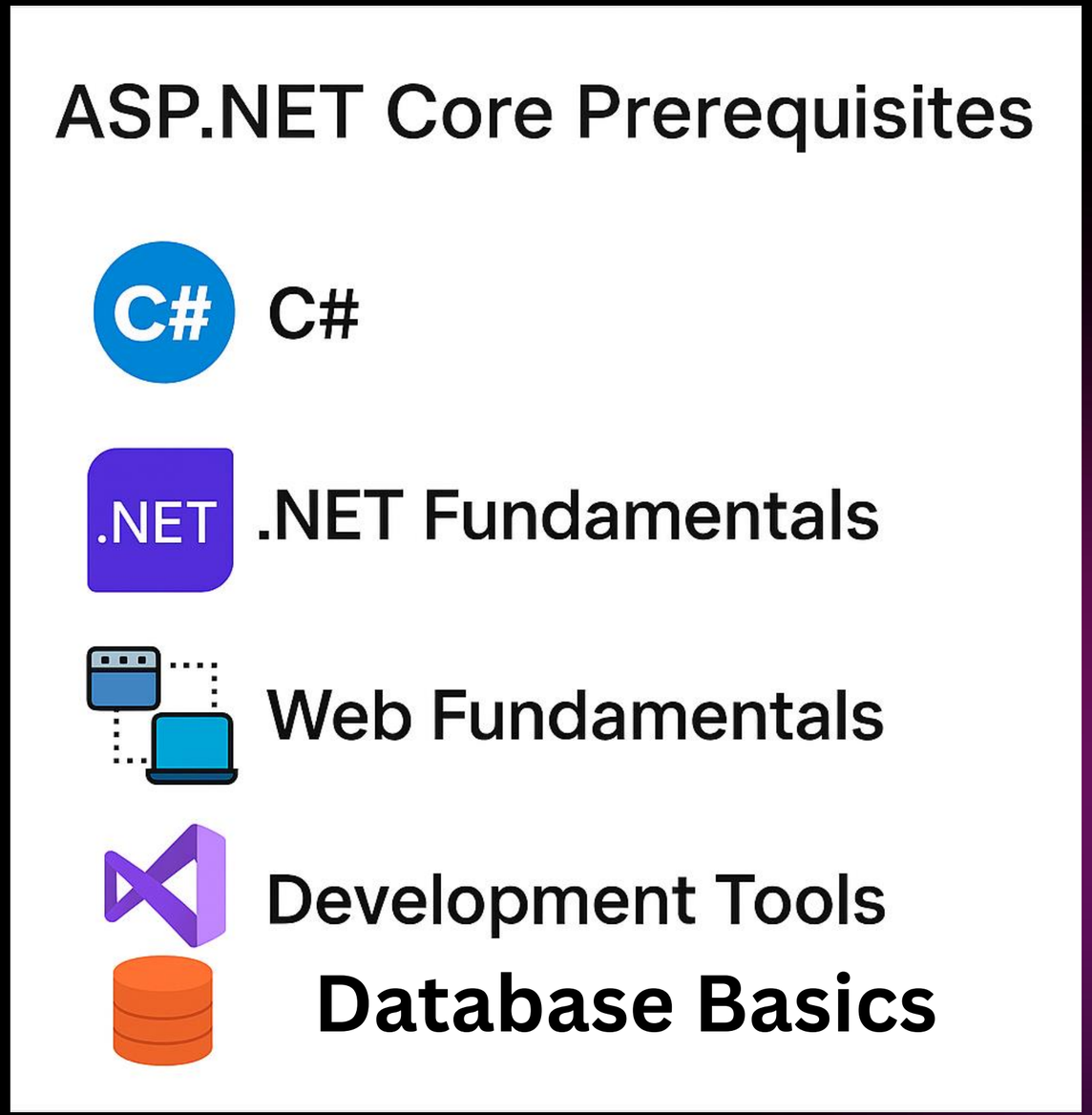
✅ **Basic Programming Knowledge**
- Understanding of C# syntax and concepts
- Object-Oriented Programming (OOP)

✅ **.NET Fundamentals**
- Familiarity with .NET Framework / .NET Core
- NuGet packages and Visual Studio basics

✅ **Web Fundamentals**
- HTTP protocol, request/response cycle
- HTML, CSS, JavaScript basics

✅ **Development Tools**
- Visual Studio / Visual Studio Code
- .NET SDK installed

✅ **Database Basics**
- SQL and relational database concepts
- Entity Framework



Basics of Web Technologies

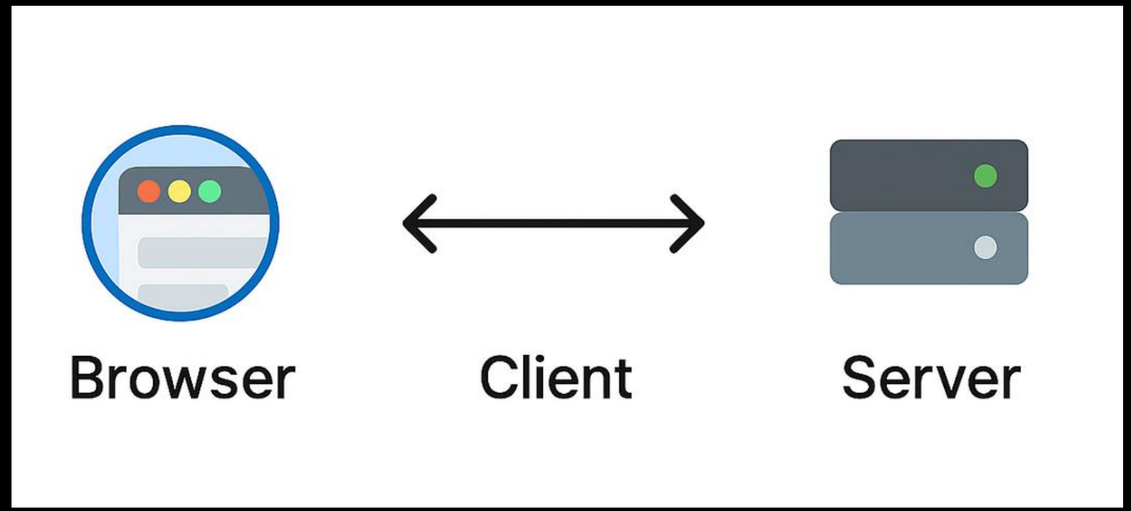
Browser & Client-Server Model

🌐 Browser

- A software used to access and display web pages.
- Sends requests to web servers and renders responses.

🔁 Client-Server Model

- Client: Sends request (browser)
- Server: Processes request and sends response



Basics of Web Technologies

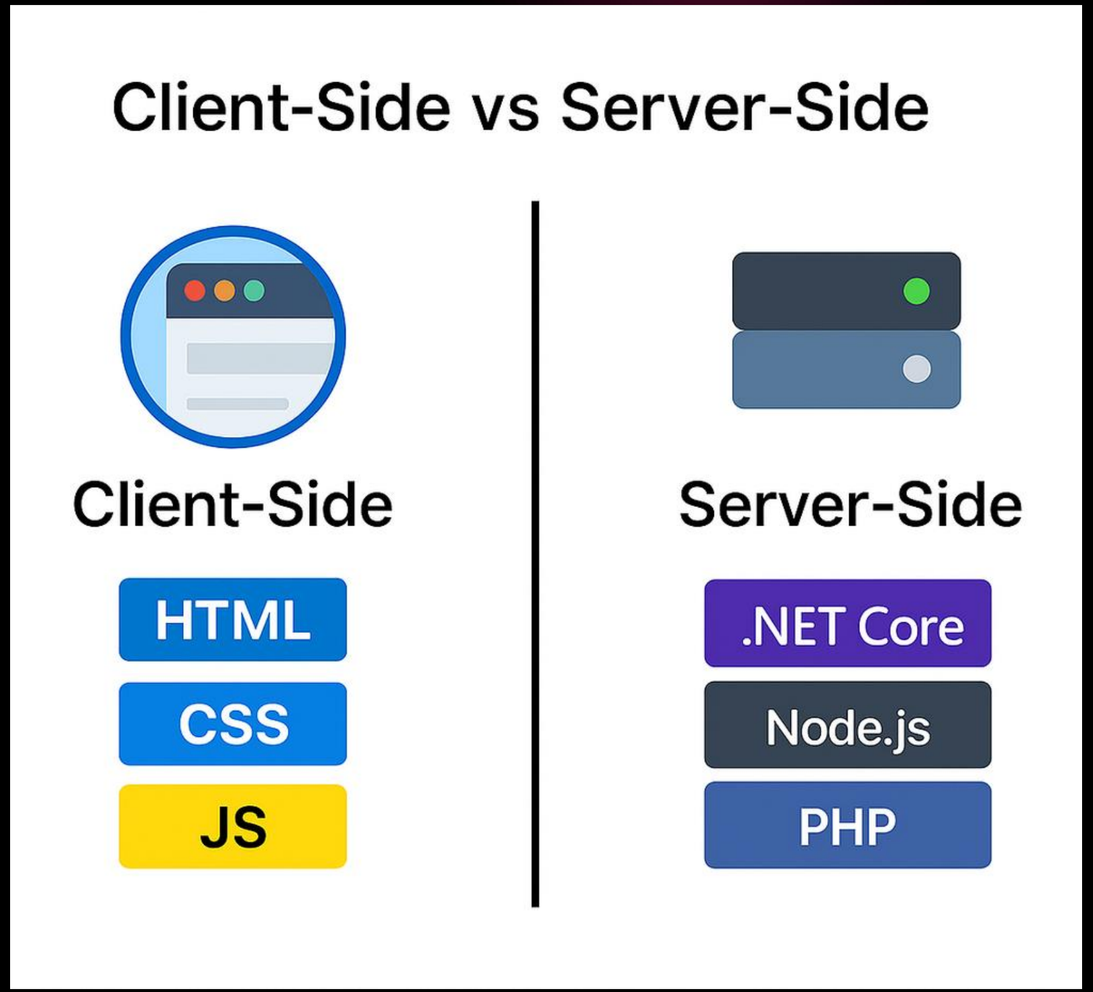
Client-Side vs Server-Side

🖥️ Client-Side

- Runs in the browser
- Technologies: HTML, CSS, JavaScript
- Handles UI and user interaction

🖧 Server-Side

- Runs on the server
- Technologies: ASP.NET Core, Node.js, PHP
- Handles logic, database, and APIs



Basics of Web Technologies

Backend & Web Technologies

⬅️ Backend
- Manages logic, data, and server-side operations
- Connects to databases and APIs

⚙️ Client-Side Technologies
- HTML, CSS, JavaScript

⚙️ Server-Side Technologies
- ASP.NET Core, Node.js, Django

🖥️ Web Servers
- IIS (Internet Information Services), Apache, Nginx

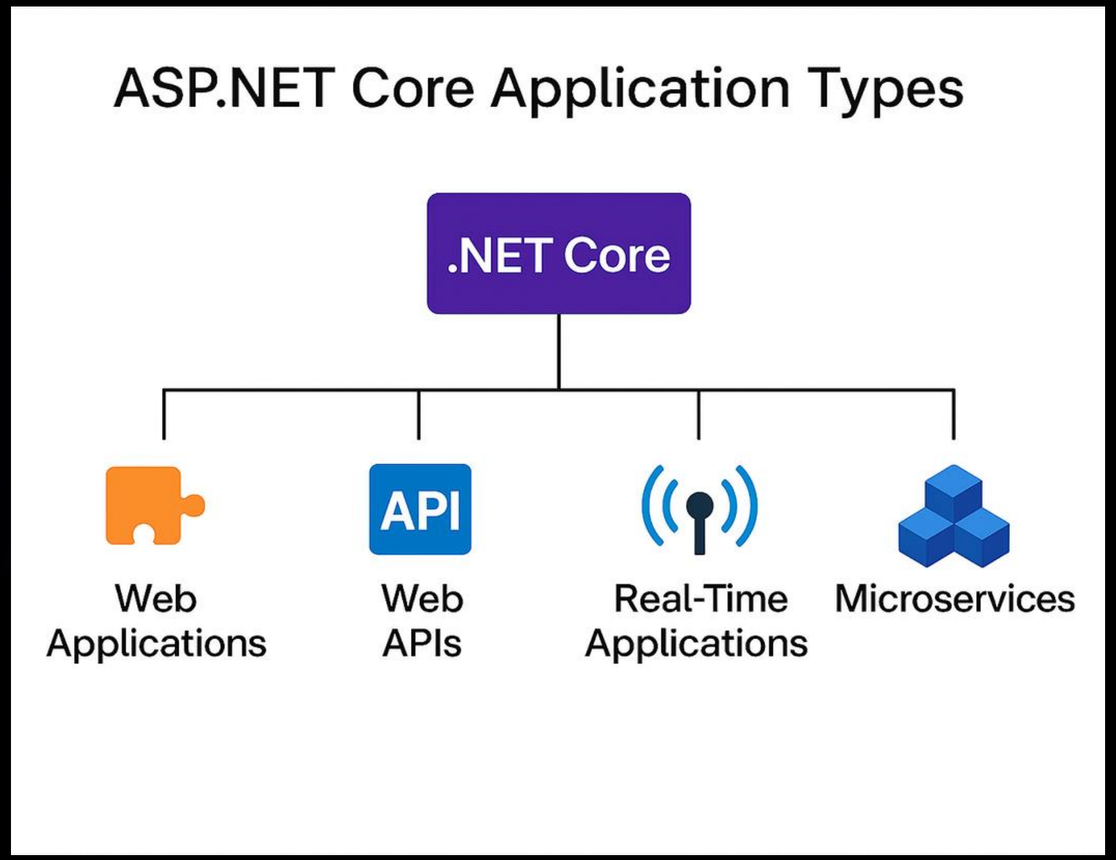
ASP.NET Core Application Types

🧩 Web Applications
- Traditional websites with dynamic content
- Built using MVC or Razor Pages

🔗 Web APIs
- RESTful services for data exchange
- Used in mobile apps, SPAs, and microservices

📡 Real-Time Applications
- Live updates using SignalR
- Examples: Chat apps, dashboards

🧱 Microservices
- Small, independent services
- Scalable and maintainable architecture



Why Use ASP.NET Core

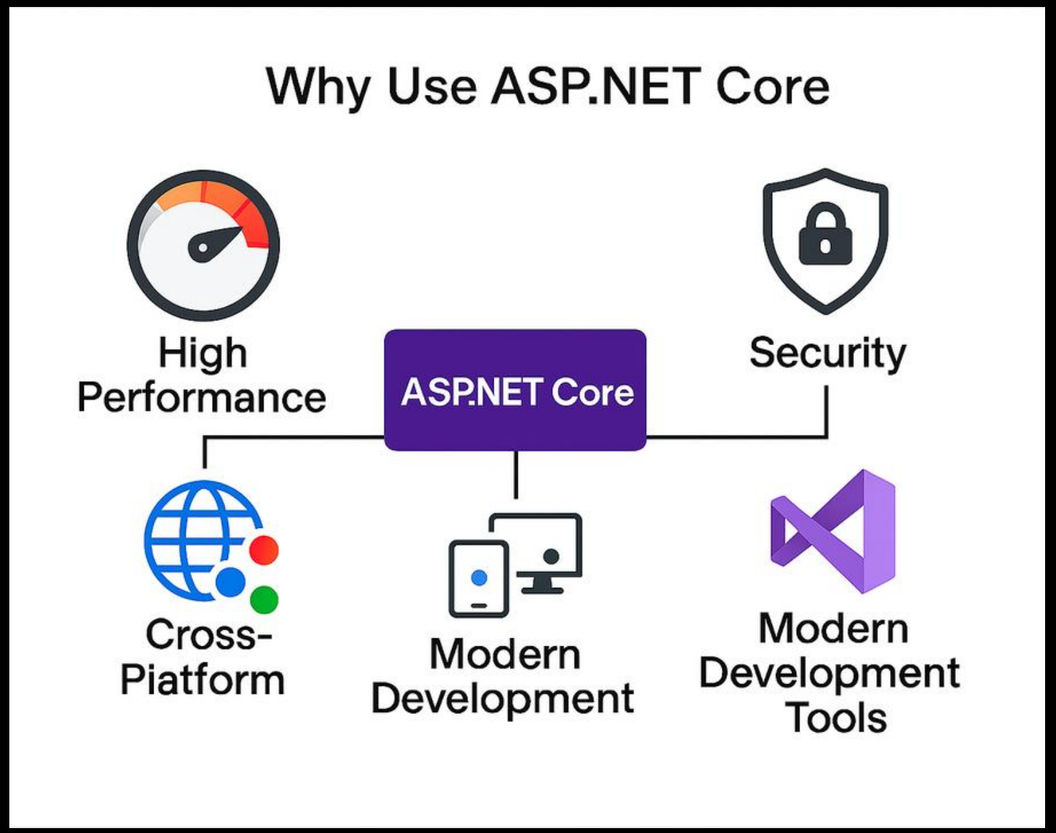
🚀 High Performance
- Fast and lightweight runtime
- Optimized for cloud and web workloads

🌍 Cross-Platform
- Runs on Windows, Linux, and macOS
- Ideal for containerized and cloud-native apps

🔒 Security
- Built-in authentication and authorization
- Regular updates and support from Microsoft

🧩 Modular & Flexible
- Use only what you need via NuGet packages
- Supports microservices and modern architectures

🛠️ Modern Development Tools
- Integrated with Visual Studio
- Supports hot reload, CLI tools, and debugging



ASP.NET Core Project Structure

📁 **Main Folders**

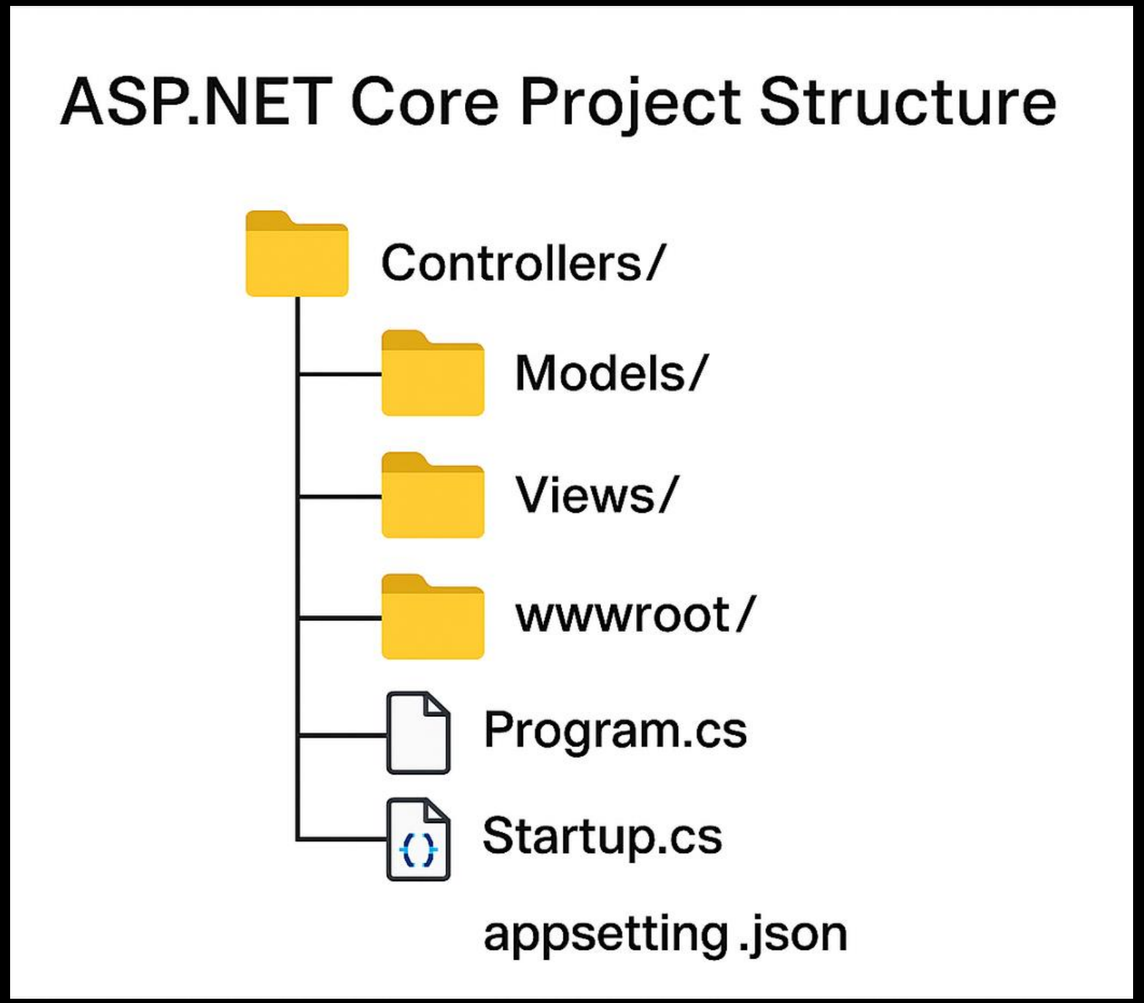
- Controllers/ – Handles incoming requests
- Models/ – Represents data and business logic
- Views/ – UI templates (for MVC apps)
- wwwroot/ – Static files (CSS, JS, images)

⚙️ **Important Files**

- Program.cs – Entry point of the application
- Startup.cs (or builder in Program.cs) Configures services and middleware
- appsettings.json – Configuration settings

🧩 **Dependencies**

- Managed via *.csproj file
- NuGet packages for additional features



MVC Architecture

🧩 Model-View-Controller (MVC) Pattern

- A design pattern used to separate concerns in web applications.

📦 Model

- Represents the data and business logic.
- Communicates with the database.

🎨 View

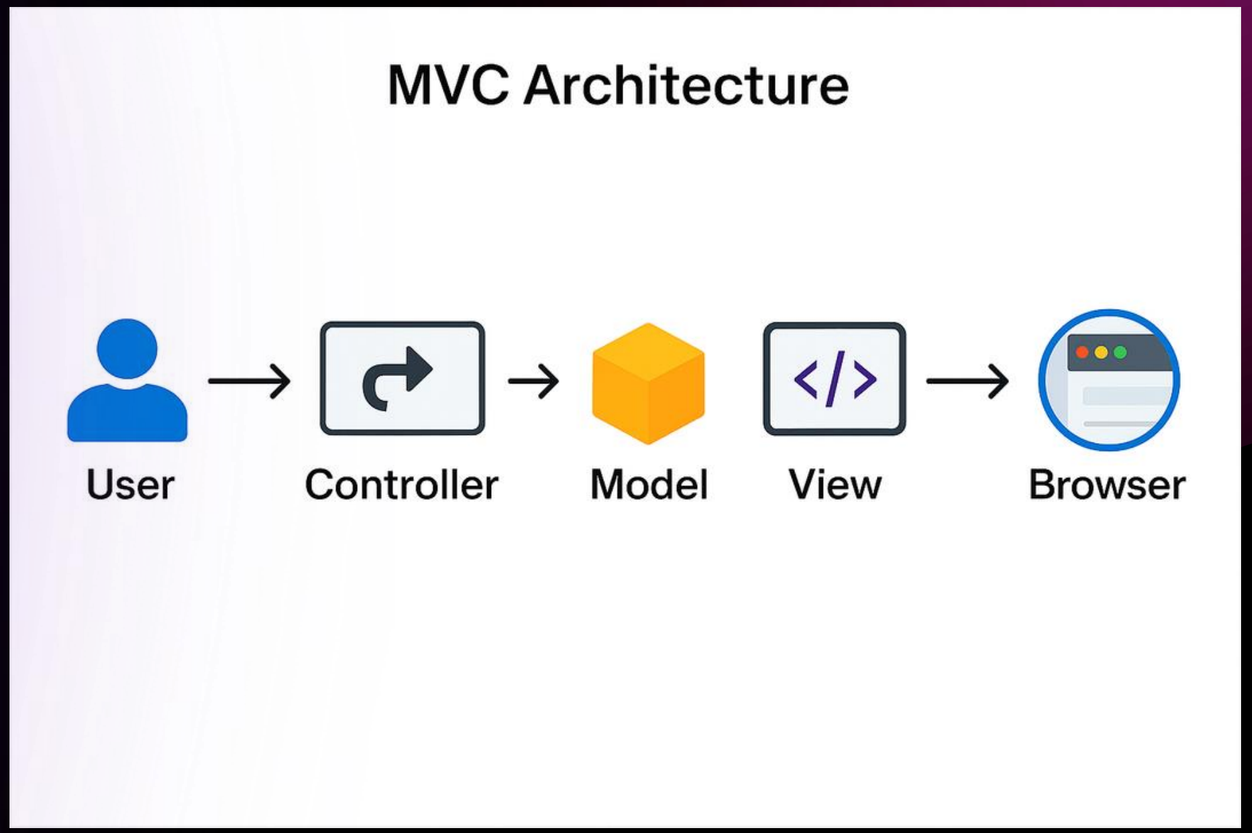
- Handles the UI and presentation.
- Uses Razor syntax to render dynamic content.

🧭 Controller

- Manages user input and application flow.
- Calls models and returns views or data.

🔄 Flow

- User → Controller → Model → View → Response

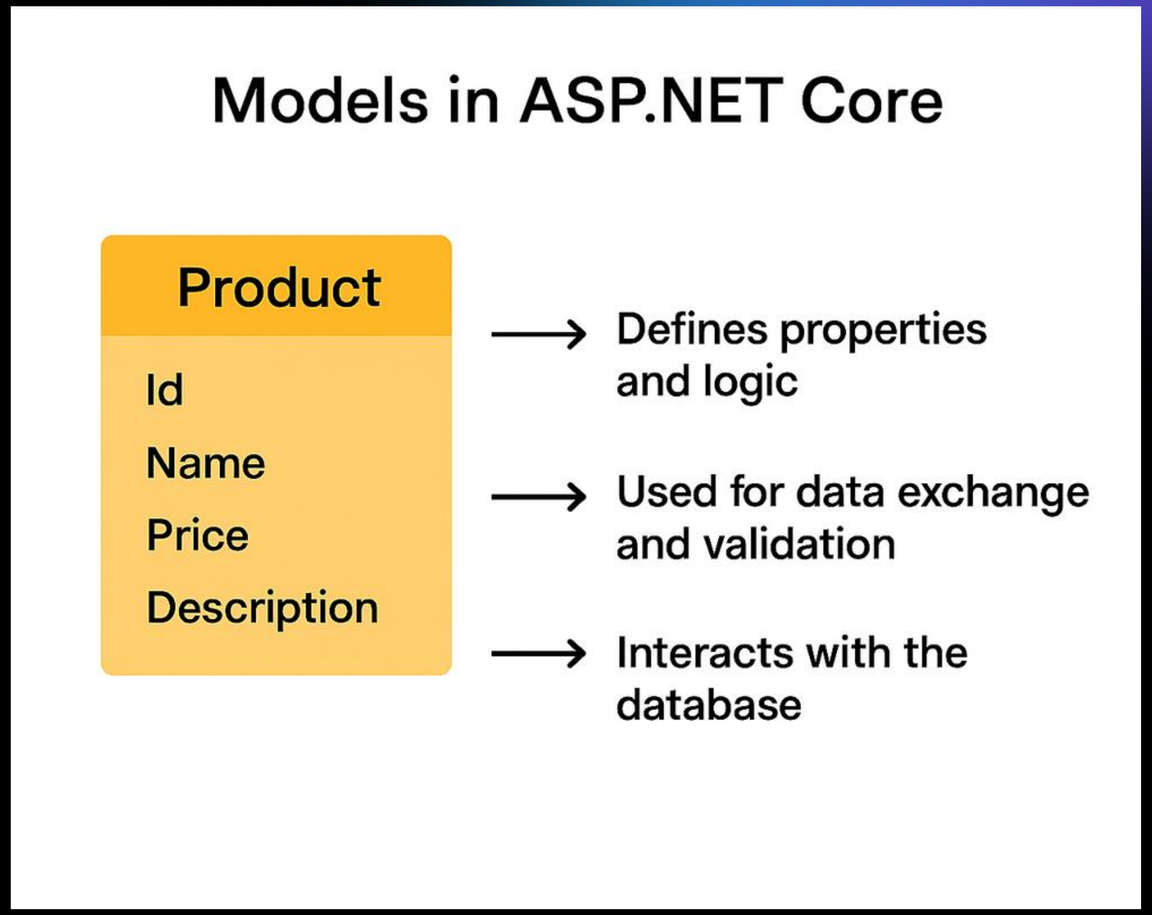


Models in ASP.NET Core

📦 What is a Model?

- Represents the data and business logic of the application.
- Used to define properties and validation rules.
- Communicates with the database via Entity Framework.

```
public class Product
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Range(0, 10000)]
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

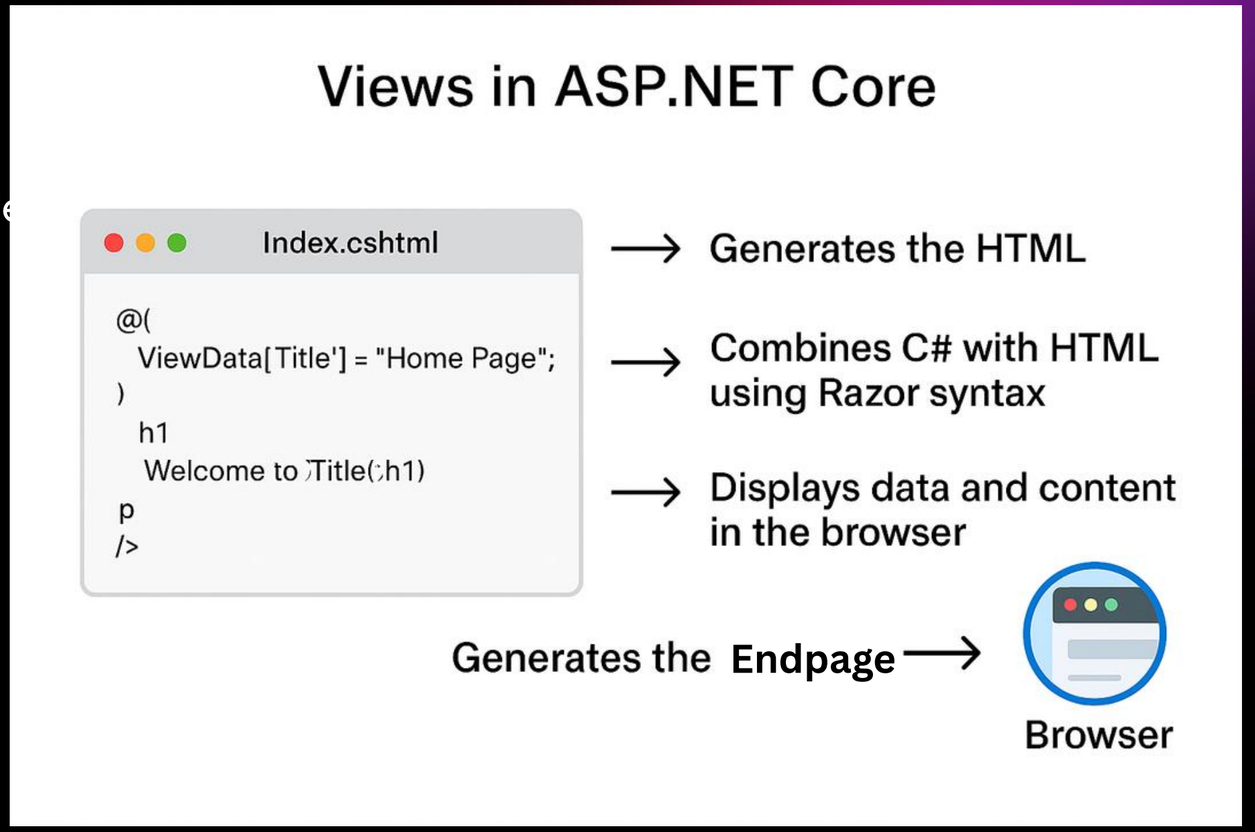


Views – Structure Overview

🎨 What is a View?

- Responsible for rendering the UI.
- Uses Razor syntax to combine HTML with C#.
- Located in the Views/ folder, organized by controlle

```
Views/
├── Home/
│   └── Index.cshtml
├── Shared/
│   └── _Layout.cshtml
```



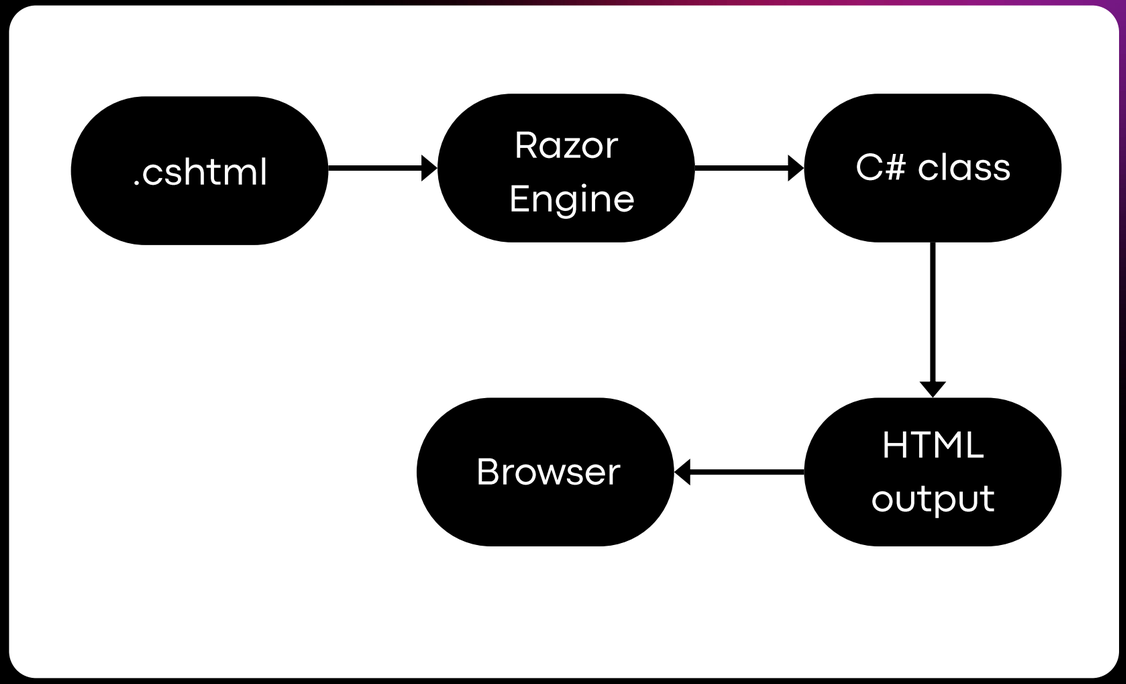
Views – Razor & Razor Engine

🧠 **What is Razor?**

- Razor is a markup syntax used in ASP.NET Core to embed C# code into HTML.
- It allows dynamic content rendering in views.
- Razor files have the .cshtml extension.

⚙️ **Razor Engine**

- The Razor Engine processes .cshtml files.
- It compiles Razor markup into C# classes.
- These classes generate HTML output sent to the browser.



Views – Razor Syntax

Basic Example

```
@{
  ViewData["Title"] = "Home Page";
}

<h1>@ViewData["Title"]</h1>
<p>Welcome to ASP.NET Core!</p>
```

With C# Logic

```
@{
  var products = new List<string> { "Laptop",
"Phone", "Tablet" };
}

<ul>
@foreach (var item in products)
{
  <li>@item</li>
}
</ul>
```

Controllers in ASP.NET Core MVC

🧩 What is a Controller?

- A Controller is a C# class that handles incoming HTTP requests.
- It acts as the intermediary between the View (UI) and the Model (data).
- Controllers are part of the MVC architecture:
- Model → View ← Controller

📁 Location & Naming

- Stored in the Controllers folder.
- Naming convention: ends with Controller (e.g., HomeController, ProductController).
- Inherits from Controller or ControllerBase.

🧠 Responsibilities

- Handle user input.
- Interact with models to retrieve or update data.
- Return appropriate responses (views, JSON, redirects, etc.)

📜 Rules: For the Action Methods

- Must be public.
- Cannot be static.
- Cannot be overloaded only by parameter type.

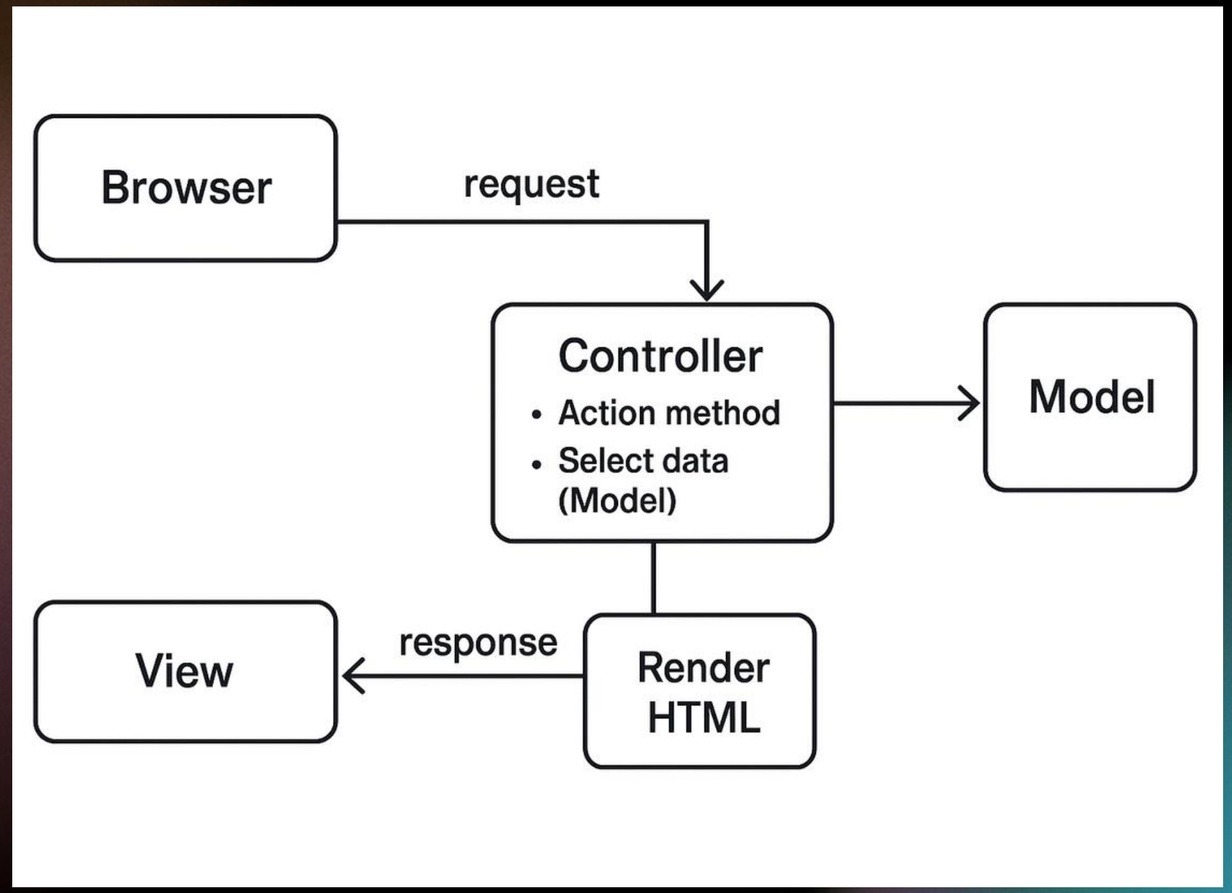
How Controllers Work

🔄 Request Lifecycle

1. 🌐 User sends a request via browser (e.g., /products/details/1).
2. 🛣️ Routing system matches the URL to a controller and action.
3. 🧠 Controller processes the request:
   - Validates input
   - Calls services or models
   - Prepares data for the view
4. 📮 Returns a response:
   - HTML view
   - JSON data
   - Redirect
   - Status code

🧪 Example Flow

- URL: /Product/Details/1
- Routed to: ProductController → Details(int id)
- Returns: View(productDetails)



Controller Code Structure

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View(); // returns ViewResult
    }

    public IActionResult GetJson()
    {
        var data = new { Name = "Rohan", Role = "Developer" };
        return Json(data); // returns JsonResult
    }

    public IActionResult RedirectToGoogle()
    {
        return Redirect("https://www.google.com"); // RedirectResult
    }
}
```

Common Action Results

Action Result	Description
 ViewResult	Returns a view (HTML page)
 JsonResult	Returns JSON data
 RedirectTo ActionResuult	Redirects to another action
 ContentResult	Returns plain text
 FileResult	Returns a downloadable file
200 StatusCodeResult	Returns HTTP status codes

Passing Data to Views - ViewData

🧩 What is ViewData?

- Definition: A dictionary of key-value pairs (ViewDataDictionary) for passing data from a controller to a view.

- Type: ViewDataDictionary (stores data as object)
- A dynamic wrapper around ViewData. Easier syntax, no casting required. Limited to the current request.

- Pros: Simple to use, works with loosely typed data.
- Cons: Requires type casting, error-prone if keys are misspelled.

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

```
@ViewData["Message"]
```

Passing Data to Views - ViewBag

👜 What is ViewBag?

- Definition: A dynamic wrapper around ViewData that allows property-like access.

- Type: dynamic (runtime binding)

- **Pros**: Cleaner syntax than ViewData.
- **Cons**: Loosely typed, no compile-time checking.

```
ViewBag.UserName = "Rohan";
```

```
@ViewBag.UserName
```

Passing Data to Views - TempData

🔄 TempData

- **Definition**: Used to store data between two requests. Data is stored in session or cookies temporarily.

- **Type**: TempDataDictionary.

- **Pros**: Useful for redirect scenarios (e.g., after form submission).
- **Cons**: Data persists only for one request; must be type-cast.

```
TempData["Message"] = "Data for next
request";
return RedirectToAction("NextAction");
```

```
var message = TempData["Message"];
```

Passing Data to Views - Strongly Typed View

📃 Strongly Typed View

- **Definition: Passing a model object directly to the view for strongly-typed access.**
- **Pros: Strongly typed access, IntelliSense support.**
- **Cons: Requires model class and binding.**

```
public class Student {
  public int Id { get; set; }
  public string Name { get; set; }
}
```

```
public IActionResult Index() {
  var student = new Student { Id = 1, Name =
"John" };
  return View(student);
}
```

```
@model Student
<p>@Model.Name</p
```

Passing Data to Views - Comparison Table

Feature	Scope	Lifetime	Type Safety	Typical Use
ViewData	Controller → View	Single request	❌	Small data, no model
ViewBag	Controller → View	Single request	❌	Quick passing of values
TempData	Across requests	One request	❌	Redirect scenarios
Strongly Typed View	Controller → View	Single request	✅	Full model data

Tag Helpers

🧩 Definition:

- Tag Helpers are server-side components that enable HTML-like syntax for Razor code.
- They make Razor views cleaner, more readable, and intelliSense-friendly.

🛠️ Key Features:

- Work like HTML tags but are powered by C#.
- Automatically bind to model properties.
- Help with form generation, validation, links, scripts, etc.

📦 Common Tag Helpers:

- <form asp-action="ActionName">
- <input asp-for="PropertyName" />
- <label asp-for="PropertyName" />
- <select asp-for="PropertyName" asp-items="Model.Options" />

✅ **Benefits:**

- Automatically binds form fields to model properties.
- Reduces manual HTML and JavaScript.
- Supports validation and model binding out of the box.

Tag Helpers - Example

```
@model Student

<form asp-action="Submit">
  <div>
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div>
    <label asp-for="Age"></label>
    <input asp-for="Age" class="form-control" />
  </div>
  <button type="submit">Submit</button>
</form>
```

```
public class Student {
  public string Name { get; set; }
  public int Age { get; set; }
}
```

Summary

👷 Prerequisites: C#, .NET basics, Visual Studio
🌐 Web Basics: Browser, Client/Server side, Backend, Technologies
🏗️ App Types: Web API, Web Apps, Real-time apps, Microservices
🚀 Why ASP.NET Core?: Fast, cross-platform, scalable, DI, unified model
📁 Project Structure: Controllers, Models, Views, wwwroot, Startup.cs
🧩 MVC Pattern: Model (data), View (UI), Controller (logic)
👩‍🎓 Models: Class structure + sample code
🖼️ Views: Razor syntax, dynamic rendering, sample code
🧭 Controllers: Action methods, IActionResult types, sample code
📩 Passing Data: ViewData, ViewBag, TempData, Strongly Typed Views + comparison
🏷️ Tag Helpers: Razor-enhanced HTML for forms, model binding, clean syntax

Thanks You!

Presentation by Rohan Gope