100 Essential C# Interview Questions and Answers

MUHAMMAD AFZAL
Web Developer | .NET Core | C#

Click To Follow

1. What is C# and how is it different from other programming languages?

Answer:
C# is a modern, object-oriented programming language developed by Microsoft. It is widely used for developing web applications, desktop applications, and game development (via Unity). It runs on the .NET framework, which provides a collection of libraries and tools to build applications efficiently.

Differences from other languages:

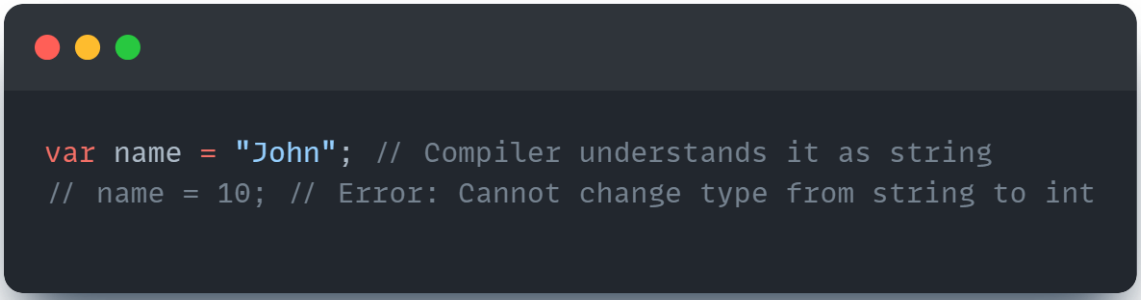
- **C# vs C++**: C# has automatic memory management (Garbage Collection) and is easier to use, while C++ gives developers more control over memory but is more complex.
- **C# vs Java**: Both are object-oriented, but C# has additional features like properties, delegates, and LINQ. Java is platform-independent due to the JVM, whereas C# runs primarily on Windows but also supports cross-platform development via .NET Core.
- **C# vs Python**: C# is statically typed (variable types are checked at compile-time), while Python is dynamically typed (types are determined at runtime). C# also runs faster due to ahead-of-time compilation.

2. What is the difference between var, object, and dynamic?

Answer:

1. var (Implicitly Typed Variable)

- The compiler determines the type at **compile-time**.
- Once assigned, its type cannot change.



2. object (Base Type for All Types)

- Can store any data type.
- Requires explicit casting when retrieving data.

```
object obj = "Hello";
string str = (string)obj; // Explicit casting required
```

3. dynamic (Runtime Type Determination)

- Type is determined **at runtime**.
- No casting is required.

```
dynamic data = "Test";
data = 100; // Allowed, as the type is determined at runtime
```

Key Differences:

Feature	`var`	`object`	`dynamic`
Type determined at	Compile-time	Compile-time	Runtime
Type change allowed?	❌ No	✅ Yes (with casting)	✅ Yes
Requires casting?	❌ No	✅ Yes	❌ No

3. What are value types and reference types in C#?

Answer:

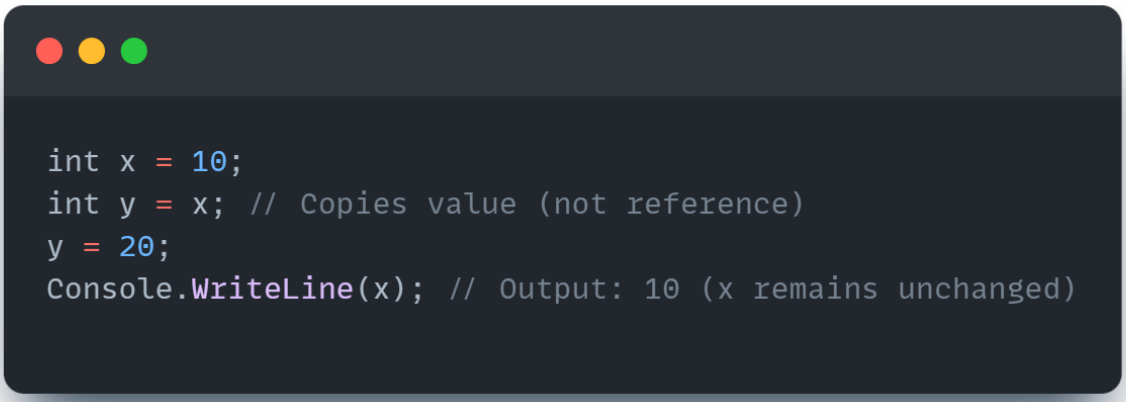
1. Value Types

- Stores the actual data in **stack memory**.

- Directly holds values, making them faster.

Examples:

`int, float, char, bool, struct, enum`



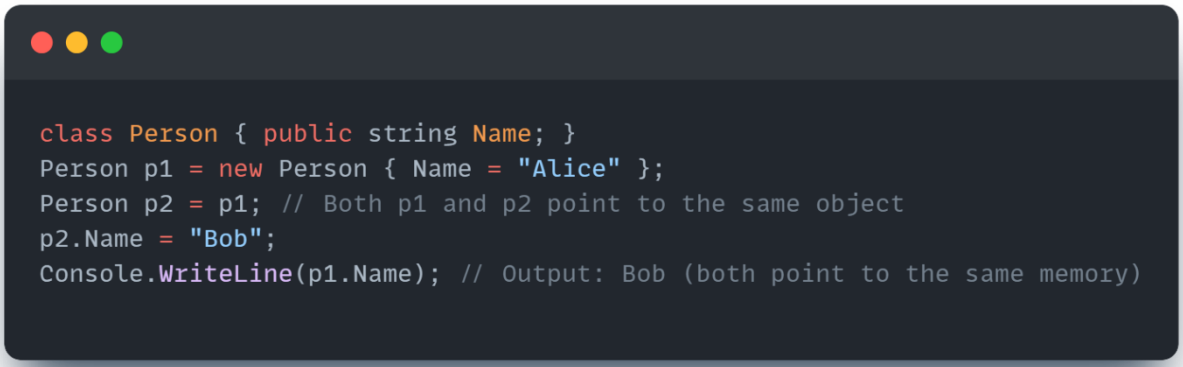
```
int x = 10;
int y = x; // Copies value (not reference)
y = 20;
Console.WriteLine(x); // Output: 10 (x remains unchanged)
```

2. Reference Types

- Stores a reference (address) in **heap memory**.
- When assigned to another variable, both point to the same memory.

Examples:

`string, class, array, object, interface, dynamic`



```
class Person { public string Name; }
Person p1 = new Person { Name = "Alice" };
Person p2 = p1; // Both p1 and p2 point to the same object
p2.Name = "Bob";
Console.WriteLine(p1.Name); // Output: Bob (both point to the same memory)
```

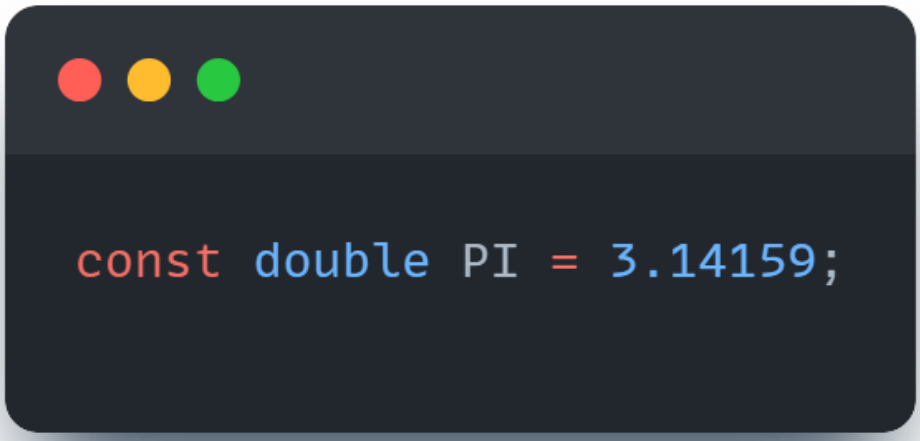
4. What is the difference between `const`, `readonly`, and `static`?

Answer:

Feature	`const`	`readonly`	`static`
Value assigned	At **compile-time**	At **runtime**	Once per class
Value change allowed?	❌ No	❌ No (after initialization)	✅ Yes (but shared across instances)
Memory location	Stored in stack	Stored in heap	Stored in heap

1. const (Constant Variable)

- Must be assigned at **compile-time**.
- Cannot be changed afterward.



```
const double PI = 3.14159;
```

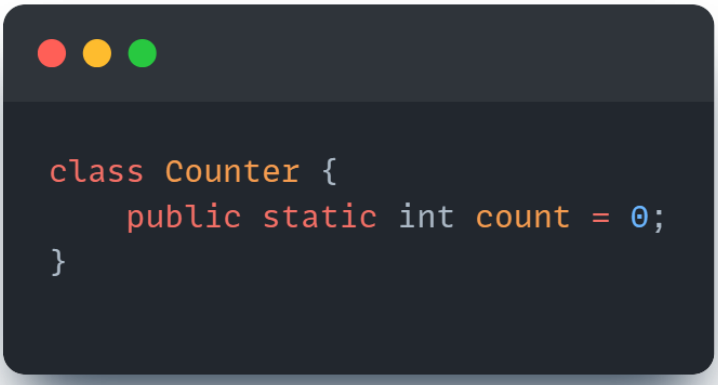
2. readonly (Runtime Constant)

- Value can be assigned **inside a constructor**, allowing different values for different objects.

```
class Car {
    public readonly int MaxSpeed;
    public Car(int speed) { MaxSpeed = speed; }
}
```

3. static (Shared Among All Instances)

- Belongs to the **class, not the instance**.
- Can be modified but affects all instances.



```
class Counter {
    public static int count = 0;
}
```

5. What is the difference between an interface and an abstract class?

Answer:

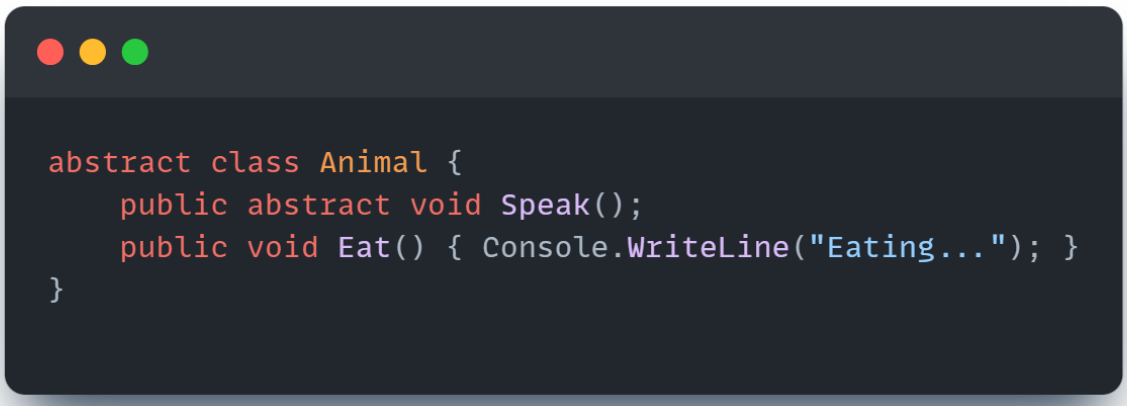
Interface

- Defines only method **signatures** (no implementation).
- Allows **multiple inheritance**.

```
interface IAnimal {
    void Speak(); // No implementation
}
```

Abstract Class

- Can have both **implemented and unimplemented** methods.
- Supports **single inheritance**.



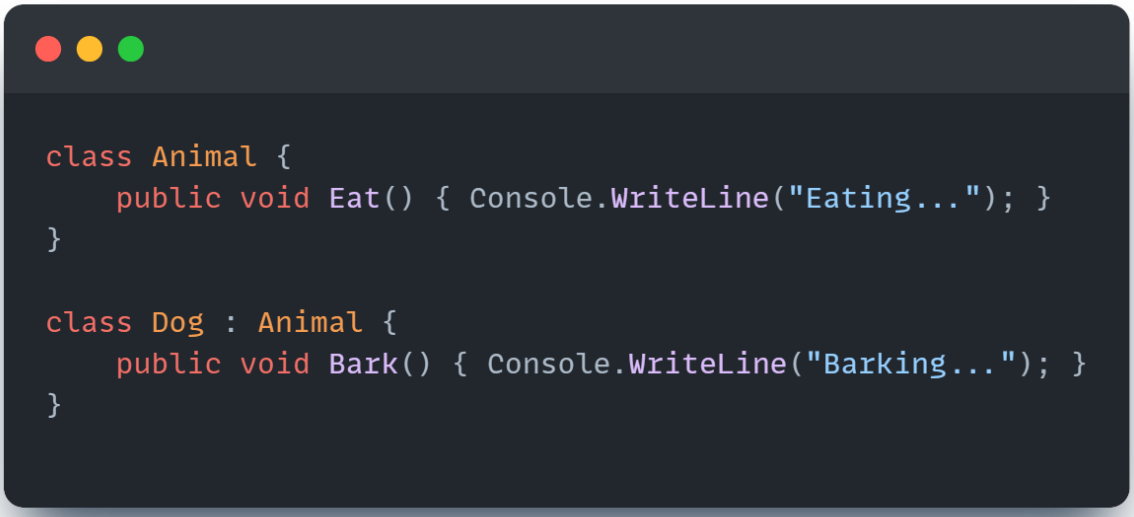
```
abstract class Animal {
    public abstract void Speak();
    public void Eat() { Console.WriteLine("Eating..."); }
}
```

Key Differences:

Feature	Interface	Abstract Class
Method Implementation	❌ Not allowed	✅ Allowed
Multiple Inheritance	✅ Yes	❌ No
Fields & Constructors	❌ Not allowed	✅ Allowed

6. Explain the concept of inheritance in C#.

Answer:
Inheritance allows one class to reuse the functionality of another class.



```
class Animal {
    public void Eat() { Console.WriteLine("Eating..."); }
}

class Dog : Animal {
    public void Bark() { Console.WriteLine("Barking..."); }
}
```

Key Features of Inheritance

- **Code Reusability**: Dog can use Eat() without rewriting.
- **Hierarchical Structure**: Parent-child relationship.

7. What is polymorphism? Provide an example.

Answer:
Polymorphism means **one method can have different behaviors** in different classes.

Method Overriding (Runtime Polymorphism)

```
class Animal {
    public virtual void Speak() { Console.WriteLine("Animal speaks"); }
}
class Dog : Animal {
    public override void Speak() { Console.WriteLine("Dog barks"); }
}
```

Method Overloading (Compile-time Polymorphism)


```
class MathOperations {
    public int Add(int a, int b) { return a + b; }
    public double Add(double a, double b) { return a + b; }
}
```

8. What are the different types of constructors in C#?

Answer:
A **constructor** is a special method that initializes an object when it is created.

Types of Constructors:

1. **Default Constructor** – No parameters, initializes default values.

```
class Car {
    public Car()
    {
        Console.WriteLine("Car created!");
    }
}
```

2. **Parameterized Constructor** – Takes arguments to initialize values.

```
class Car {
    public string Brand;
    public Car(string brand) { Brand = brand; }
}
```

3. **Copy Constructor** – Copies values from another object.

```
class Car {
    public string Brand;
    public Car(Car c) { Brand = c.Brand; }
}
```

4. **Static Constructor** – Runs once per class, used to initialize static members.

```
class Car {
    static Car() { Console.WriteLine("Static Constructor Called!"); }
}
```

5. **Private Constructor** – Prevents object instantiation outside the class.

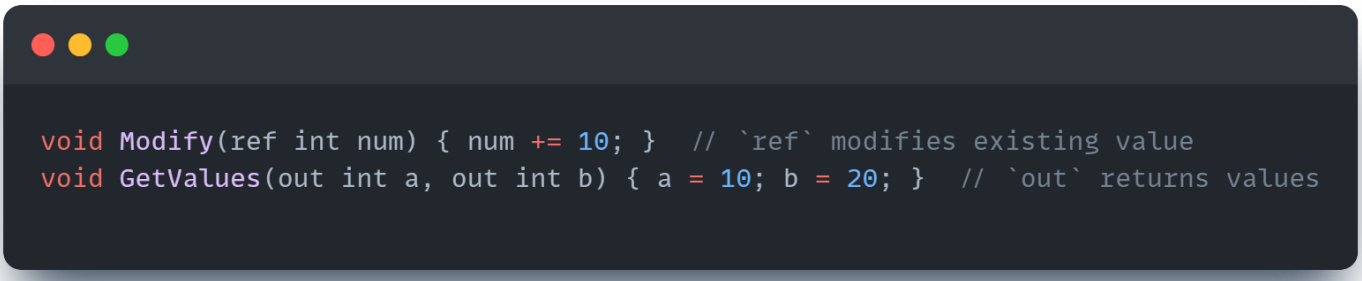
```
class Singleton {
    private Singleton() { }
}
```

9. What is the difference between ref and out parameters?

Answer:

Feature	`ref`	`out`
Initial value required?	✅ Yes	❌ No
Used for?	Modifying existing values	Returning multiple values
Example	`Calculate(ref x);`	`GetValues(out x, out y);`

Example:



```
void Modify(ref int num) { num += 10; }  // `ref` modifies existing value
void GetValues(out int a, out int b) { a = 10; b = 20; }  // `out` returns values
```

10. What is a sealed class in C#?

Answer:
A sealed class cannot be inherited. It is used to prevent modifications in derived classes.

```
sealed class Car { }  // This class cannot be inherited
```

11. Explain exception handling in C#. What are try, catch, finally, and throw?

Answer:

- `try`: The block where code execution is attempted.
- `catch`: Handles exceptions if they occur.
- `finally`: Runs after try-catch (whether an exception occurs or not).
- `throw`: Used to explicitly throw an exception.

```
try {
    int x = 10 / 0;
} catch (Exception ex) {
    Console.WriteLine("Error: " + ex.Message);
} finally {
    Console.WriteLine("Execution complete.");
}
```

12. What are tuples in C#?

Answer:
A **tuple** is a lightweight data structure that holds multiple values.

```
var person = (Name: "John", Age: 30);
Console.WriteLine(person.Name);   // Output: John
```

13. What is the difference between == and .Equals()?

Answer:

Operator	Purpose
`==`	Compares **values** (for value types) and **references** (for objects)
`.Equals()`	Compares only **values**, even for reference types

```
string a = "hello";
string b = new string("hello");

Console.WriteLine(a == b);      // True (value comparison)
Console.WriteLine(a.Equals(b)); // True (value comparison)
```

14. What is a nullable type in C#?

Answer:
A **nullable type** allows value types (int, bool, etc.) to have null values.

```
int? age = null; // Can hold a null value
```

15. How does garbage collection work in C#?

Answer:
Garbage Collection (GC) **automatically removes unused objects** from memory.

GC Runs in 3 Phases:

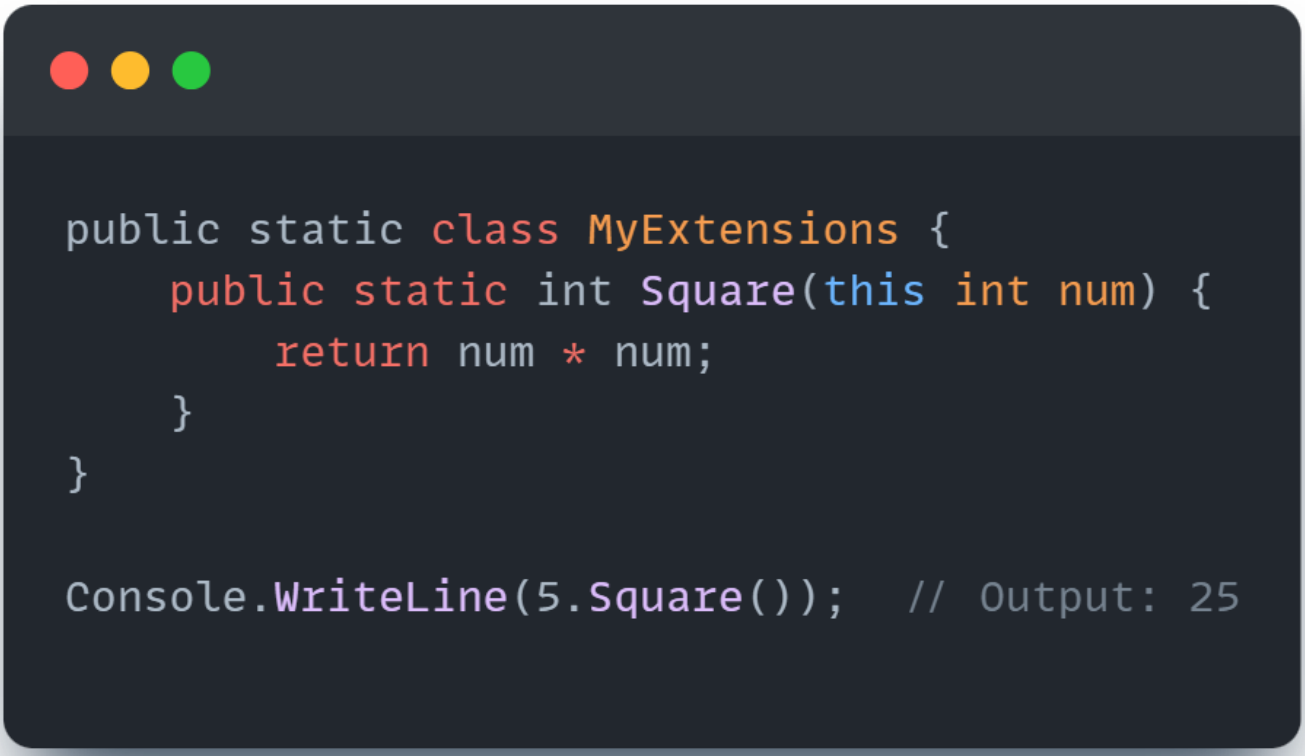
1. **Mark** – Identifies unused objects.
2. **Sweep** – Removes them.
3. **Compact** – Reorganizes memory.

To trigger GC manually:

```
GC.Collect();
```

16. What are extension methods?

Answer:
Extension methods allow adding new methods to existing types **without modifying them**.



```
public static class MyExtensions {
    public static int Square(this int num) {
        return num * num;
    }
}

Console.WriteLine(5.Square());  // Output: 25
```

17. What is the difference between Array and List<T>?

Answer:

Feature	`Array`	`List<T>`
Size	Fixed	Dynamic
Performance	Faster	Slightly slower (due to resizing)

```
int[] arr = new int[3];     // Fixed size
List<int> list = new List<int>(); // Dynamic size
```

18. What is a delegate in C#?

Answer:
A delegate is a **type-safe function pointer** that refers to methods.

```
delegate void MyDelegate(string message);

void Display(string msg) { Console.WriteLine(msg); }

MyDelegate del = Display;
del("Hello");
```

19. What is an event in C#?

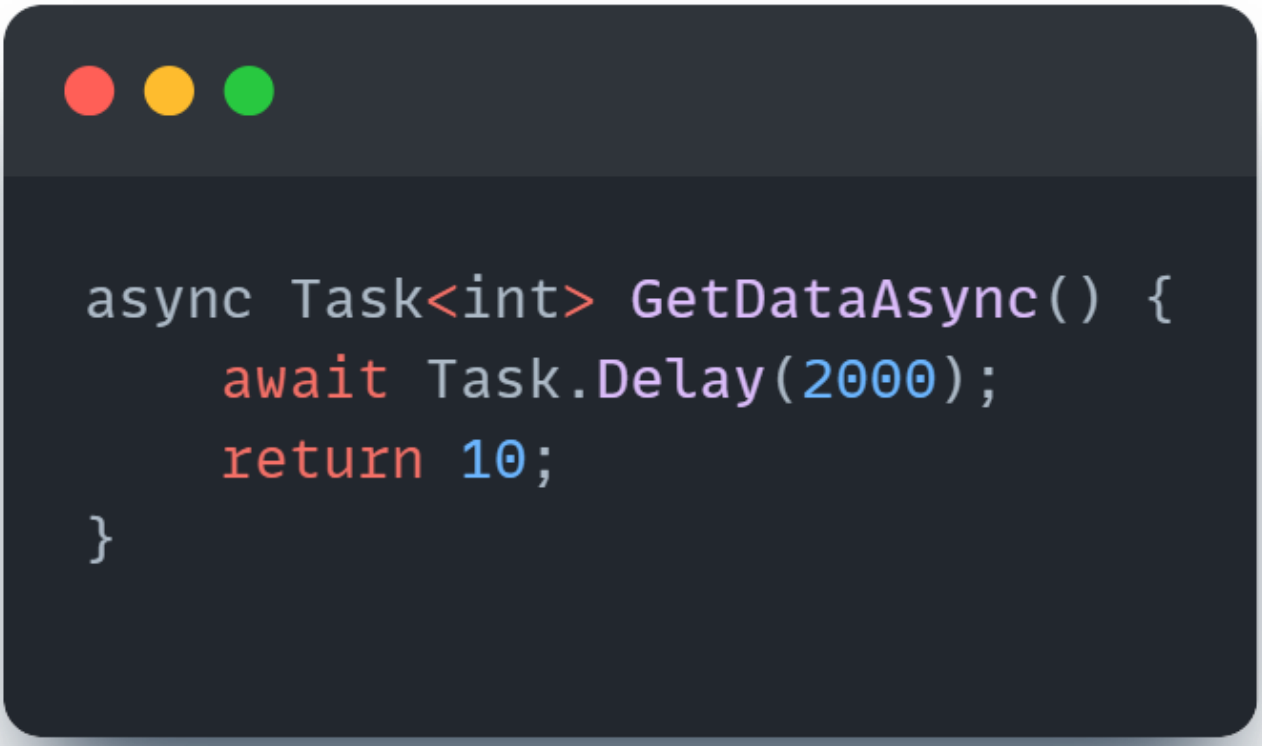
Answer:
An **event** is a special delegate that follows the **publisher-subscriber model**.

```
class Button {
    public event Action Clicked;
    public void Click() { Clicked?.Invoke(); }
}

Button btn = new Button();
btn.Clicked += () => Console.WriteLine("Button Clicked!");
btn.Click();
```

20. Explain async and await in C#.

Answer:
They enable **asynchronous programming** to improve performance.



```
async Task<int> GetDataAsync() {
    await Task.Delay(2000);
    return 10;
}
```

How it Works:

- `async` – Marks a method as asynchronous.
- `await` – Waits for a task to complete **without blocking** execution.

21. What are the four pillars of OOP?

Answer:
The four main principles of Object-Oriented Programming (OOP) are:

1. **Encapsulation** – Hides data and exposes only necessary parts.
2. **Abstraction** – Shows only essential details and hides the complexity.
3. **Inheritance** – Allows a class to reuse properties and methods from another class.
4. **Polymorphism** – Allows methods to behave differently based on the object type.

22. Explain encapsulation with an example.

Answer:
Encapsulation **restricts direct access to class data** and allows controlled access using methods (getters/setters).

```
class BankAccount {
    private double balance;  // Private field (hidden)

    public void Deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    public double GetBalance() {
        return balance;  // Controlled access
    }
}
```

Here, `balance` is **hidden** from direct access, and only the Deposit() and GetBalance() methods control how it's modified.

23. How does method overloading differ from method overriding?

Answer:

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with **same name but different parameters**	Child class modifies a method from the parent class
Return Type	Can be different	Must be the same
Parameters	Must be different	Must be the same
Modifier	No `virtual` or `override`	Uses `virtual` in base class and `override` in child class

Example:

```
// Method Overloading
class MathOperations {
    public int Add(int a, int b) { return a + b; }
    public double Add(double a, double b) { return a + b; }
}

// Method Overriding
class Parent {
    public virtual void Show() { Console.WriteLine("Parent Show"); }
}

class Child : Parent {
    public override void Show() { Console.WriteLine("Child Show"); }
}
```

24. What is an abstract method?

Answer:
An **abstract method** is a method **without implementation** in the base class, forcing child classes to define it.

```
abstract class Animal {
    public abstract void MakeSound(); // No body
}

class Dog : Animal {
    public override void MakeSound() { Console.WriteLine("Woof!"); }
}
```

25. Explain the base and this keywords.

Answer:

- base – Refers to the **parent class**. Used to call base class methods or constructors.
- this – Refers to **the current class instance**.

Example:

```
class Parent {
    public void Show() { Console.WriteLine("Parent Show"); }
}

class Child : Parent {
    public void Display() {
        base.Show();  // Calls Parent's Show()
    }
}
class Car {
    private string brand;
    public Car(string brand) {
        this.brand = brand;  // Refers to the current instance
    }
}
```

26. What is dependency injection?

Answer:
Dependency Injection (DI) is a design pattern that **reduces dependencies** between classes by injecting objects instead of creating them inside the class.

Example (Constructor Injection):

```
class Engine { }

class Car {
    private Engine _engine;

    public Car(Engine engine) { // Injected via constructor
        _engine = engine;
    }
}
```

DI is widely used in ASP.NET Core to manage services efficiently.

27. Explain SOLID principles in C#.

Answer:
SOLID are five key design principles for writing clean and maintainable code.

1. **S** – **Single Responsibility Principle** (Each class should have one purpose).
2. **O** – **Open/Closed Principle** (Classes should be open for extension, but closed for modification).
3. **L** – **Liskov Substitution Principle** (Derived classes should work as substitutes for base classes).
4. **I** – **Interface Segregation Principle** (No large interfaces; split them into smaller, specific ones).
5. **D** – **Dependency Inversion Principle** (High-level modules should not depend on low-level modules).

28. What is an interface? How is it different from an abstract class?

Answer:

Feature	Interface	Abstract Class
Contains	Only method **signatures**	Can have **implemented methods**
Variables	No fields allowed	Can have fields
Multiple Inheritance	✅ Supported	❌ Not supported
Access Modifiers	Always `public`	Can be `private`, `protected`, etc.

Example:

```
interface IAnimal {
    void MakeSound();  // No implementation
}

abstract class Animal {
    public abstract void Eat();  // Abstract method
    public void Sleep() { Console.WriteLine("Sleeping"); } // Normal method
}
```

29. What is a partial class?

Answer:
A **partial class** allows splitting a class into multiple files for better organization.

Example:

```
// File1.cs
partial class MyClass {
    public void Method1() { Console.WriteLine("Method1"); }
}

// File2.cs
partial class MyClass {
    public void Method2() { Console.WriteLine("Method2"); }
}
```

30. What is a singleton pattern? How do you implement it in C#?

Answer:
A **singleton pattern** ensures that a class has **only one instance** throughout the application.

Implementation:

```
class Singleton {
    private static Singleton _instance;
    private Singleton() { }  // Private constructor

    public static Singleton GetInstance() {
        if (_instance == null) _instance = new Singleton();
        return _instance;
    }
}
```

Usage:

```
Singleton obj1 = Singleton.GetInstance();
Singleton obj2 = Singleton.GetInstance();
Console.WriteLine(obj1 == obj2); // Output: True
```

Here, `obj1` and `obj2` are the same instance.

31. What is .NET Core?

Answer:
.NET Core is a **cross-platform, open-source framework** used to develop modern web applications, APIs, and microservices. It is lightweight, modular, and runs on **Windows, Linux, and macOS**.

32. How is .NET Core different from the .NET Framework?

Answer:

Feature	.NET Core	.NET Framework
Platform	Cross-platform (Windows, Linux, macOS)	Windows-only
Open-Source	✅ Yes	❌ No
Performance	Faster and more efficient	Slower due to older architecture
Dependency Injection	Built-in	Needs third-party libraries
Hosting	Self-hosting with Kestrel	IIS-dependent

.NET Core is **modern and lightweight**, whereas .NET Framework is **Windows-focused and heavier**.

33. What is Kestrel in .NET Core?

Answer:
Kestrel is the **built-in web server** for .NET Core applications. It is **lightweight, fast, and cross-platform**.

Example:

When you run a .NET Core app, it uses Kestrel to **serve HTTP requests**.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () ⇒ "Hello, World!");
app.Run(); // Uses Kestrel server
```

34. What are the different types of hosting models in .NET Core?

Answer:
.NET Core supports **two hosting models**:

1. **In-Process Hosting**
    - Runs the app **inside IIS** for better performance.
    - Uses AspNetCoreModuleV2.
2. **Out-of-Process Hosting**
    - Runs the app **separately** using **Kestrel**, and IIS acts as a reverse proxy.

Example Configuration in web.config (In-Process):

```xml
<aspNetCore processPath="dotnet" arguments="MyApp.dll" hostingModel="InProcess" />
```

35. Explain the middleware pipeline in ASP.NET Core.

Answer:
Middleware in ASP.NET Core is a **pipeline of components** that handle HTTP requests **step by step**.

Example Middleware Pipeline in Program.cs:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Use(async (context, next) => { Console.WriteLine("Middleware 1"); await next();
});
app.Use(async (context, next) => { Console.WriteLine("Middleware 2"); await next();
});
app.MapGet("/", () => "Hello World!");
app.Run();
```

36. What is Dependency Injection (DI) in .NET Core?

Answer:
Dependency Injection (DI) is a **design pattern** that helps manage dependencies by **injecting them instead of creating them manually**.

Example:

```
// Service Interface
public interface IMyService { string GetData(); }
// Service Implementation
public class MyService : IMyService {
    public string GetData() ⇒ "Hello from MyService!"; }
// Register in DI
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<IMyService, MyService>();
var app = builder.Build();
app.Run();
```

Here, IMyService is **injected** into other components automatically.

37. What is the Startup.cs file in .NET Core?

Answer:
In **.NET 5 and earlier**, Startup.cs was used to **configure services and middleware**. It had two main methods:

1. ConfigureServices() – Registers services (like DI, logging, authentication).
2. Configure() – Defines the middleware pipeline.

Example Startup.cs:

```
public class Startup {
    public void ConfigureServices(IServiceCollection services) { services.AddControllers();
    }
    public void Configure(IApplicationBuilder app) {
    app.UseRouting();
        app.UseEndpoints(endpoints ⇒ {
            endpoints.MapControllers();
        });
    }
}
```

In .NET 6+, Startup.cs is merged into Program.cs.

38. Explain the difference between `appsettings.json` and `secrets.json`?

Answer:

Feature	appsettings.json	secrets.json
Purpose	Stores application settings (DB, logging, APIs)	Stores sensitive data (passwords, API keys)
Location	Inside the project	Outside the project (user profile)
Security	Visible in source code	**Not committed** to Git

Example `appsettings.json`:

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=MyDB;User Id=myuser;Password=mypassword;"
  }
}
```

Use secrets.json for sensitive data:

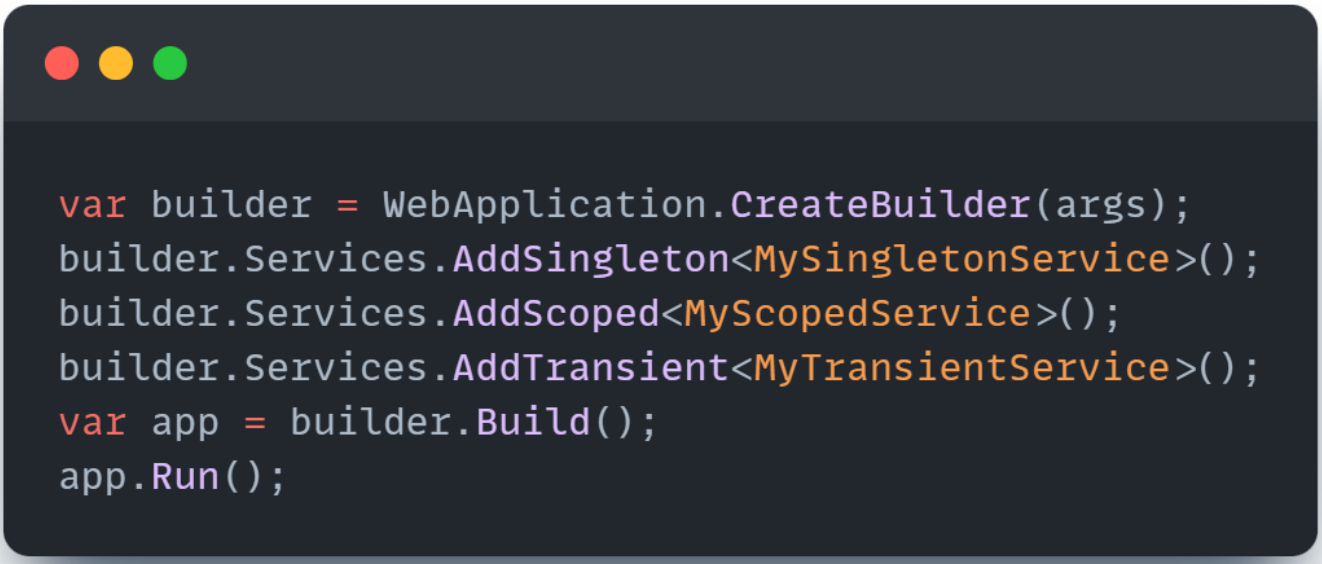
```shell
dotnet user-secrets set "ApiKey" "12345"
```

39. What are the different ways to configure services in .NET Core?

Answer:
You can register services in Program.cs using different lifetimes:

1. **Singleton** – Created **once** for the entire app.
2. **Scoped** – Created **per request**.
3. **Transient** – Created **every time it's needed**.

Example:



```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<MySingletonService>();
builder.Services.AddScoped<MyScopedService>();
builder.Services.AddTransient<MyTransientService>();
var app = builder.Build();
app.Run();
```

40. What is the role of Program.cs in .NET Core?

Answer:
In **.NET 6+**, Program.cs replaces Startup.cs and contains:

1. **Creating the application** (WebApplication.CreateBuilder(args))
2. **Configuring services** (builder.Services.AddControllers())
3. **Defining middleware** (app.UseRouting())

Example `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);
// Add services
builder.Services.AddControllers();
var app = builder.Build();
// Middleware pipeline
app.UseRouting();
app.MapControllers();
app.Run();
```

This is the entry point of the application.

41. What is MVC architecture?

Answer:
MVC (Model-View-Controller) is a design pattern used in ASP.NET Core to separate application concerns:

- **Model** – Represents the data and business logic.
- **View** – Handles the UI (HTML, CSS, etc.).
- **Controller** – Handles user requests and processes them.

Example:

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

This follows the MVC pattern where the controller (HomeController) handles a request and returns a view.

42. Explain the life cycle of an ASP.NET Core MVC request.

Answer:
The **ASP.NET Core MVC request life cycle** follows these steps:

1. **Request arrives** – A user makes a request to a controller.
2. **Routing** – The request is matched to a controller action.
3. **Model Binding** – Data from the request is bound to model objects.
4. **Action Execution** – The controller action runs and processes data.
5. **Result Execution** – The action returns a response (ViewResult, JsonResult, etc.).
6. **Response Sent** – The response is sent back to the client.

43. What are Tag Helpers in ASP.NET Core MVC?

Answer:
Tag Helpers are **server-side components** that make it easier to write HTML in Razor views.

Example:

```html
<form asp-action="Login" asp-controller="Account">
    <input asp-for="Email" />
    <button type="submit">Submit</button>
</form>
```

Here, **`asp-action`** and **`asp-for`** are Tag Helpers that generate proper form elements dynamically.

44. What is Model Binding in ASP.NET Core?

Answer:
Model Binding **automatically maps HTTP request data** (from forms, query strings, JSON) to C# objects.

Example:

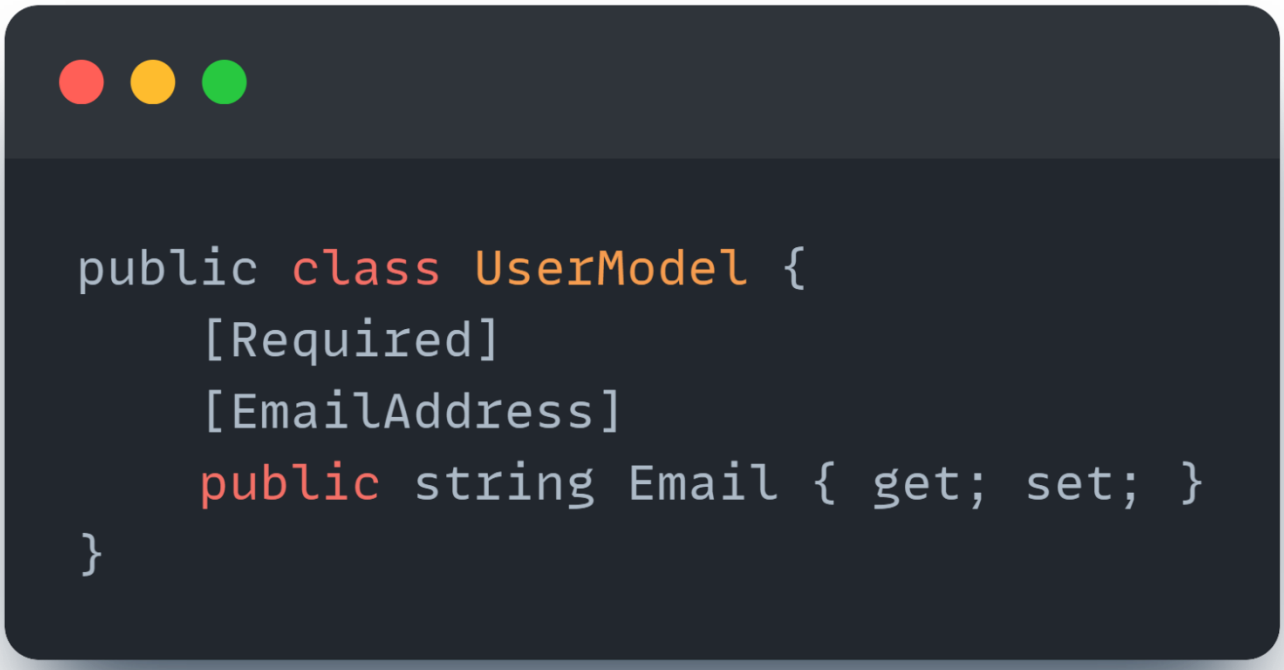
```
public IActionResult Register(UserModel model) { return View(model); }
```

If a form sends `name=John&email=john@example.com`, Model Binding **fills** the UserModel object automatically.

45. What is Model Validation in ASP.NET Core?

Answer:
Model Validation ensures that **form data is correct** before processing it.

Example:



If the email field is empty, ASP.NET Core **shows validation errors** automatically.

46. What are TempData, ViewData, and ViewBag? How do they differ?

Answer:

Feature	TempData	ViewData	ViewBag
Type	Dictionary (`TempData["Key"]`)	Dictionary (`ViewData["Key"]`)	Dynamic (`ViewBag.Key`)
Lifetime	**Persists between requests**	**Only for the current request**	**Only for the current request**
Use Case	Pass data across pages	Pass data to views	Pass data to views

Example:

Untitled-1

```
TempData["Message"] = "Saved successfully!";
ViewData["Title"] = "Home Page";
ViewBag.User = "John Doe";
```

47. How do you handle exceptions in ASP.NET Core MVC?

Answer:
Exception handling is done using **Middleware**, try-catch, or custom error pages.

Example Using Middleware (Program.cs):

Untitled-1

```
app.UseExceptionHandler("/Home/Error");
app.UseStatusCodePagesWithRedirects("/Home/Error?code={0}");
```

Example Using `try-catch` in a Controller:

Untitled-1

```
public IActionResult Index() {
    try {
        int x = 0;
        int y = 10 / x; // This will throw an error
    } catch (Exception ex) {
        return View("Error", ex.Message);
    }
    return View();
}
```

48. What is Routing in ASP.NET Core?

Answer:
Routing maps URLs to controller actions.

Example: Default Route in Program.cs

Untitled-1

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

If a user visits `/Products/Details/1`, it calls:
ProductsController -> `Details(int id)`

49. Explain how Action Filters work in ASP.NET Core.

Answer:
Action Filters execute code before or after controller actions.

Example: Creating a Custom Action Filter

```
Untitled-1

public class LogActionFilter : ActionFilterAttribute {
    public override void OnActionExecuting(ActionExecutingContext context) {
        Console.WriteLine("Action is starting...");
    }
}
```

Using It in a Controller

```
Untitled-1

[LogActionFilter]
public IActionResult Index() {
    return View();
}
```

This logs a message before the action runs.

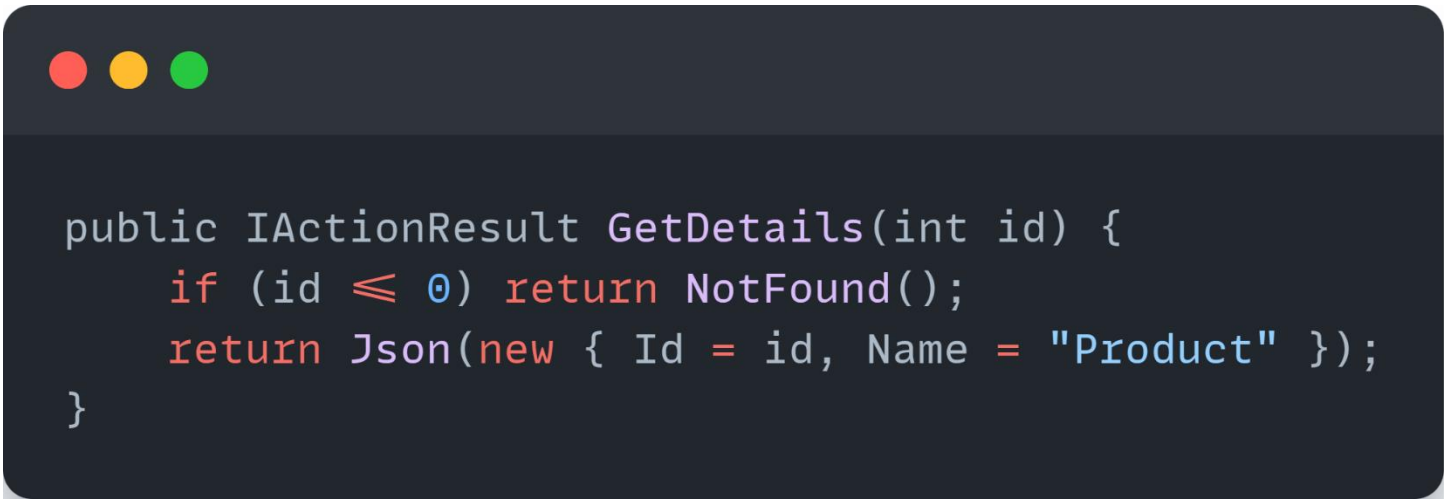
50. What is the purpose of IActionResult in ASP.NET Core?

Answer:
IActionResult is an **interface** that allows controllers to return **different types of responses**.

Common IActionResult Types

Type	Example
`ViewResult`	`return View();`
`JsonResult`	`return Json(new { name = "John" });`
`RedirectResult`	`return Redirect("/Home/Index");`
`ContentResult`	`return Content("Hello World");`
`NotFoundResult`	`return NotFound();`

Example in a Controller:

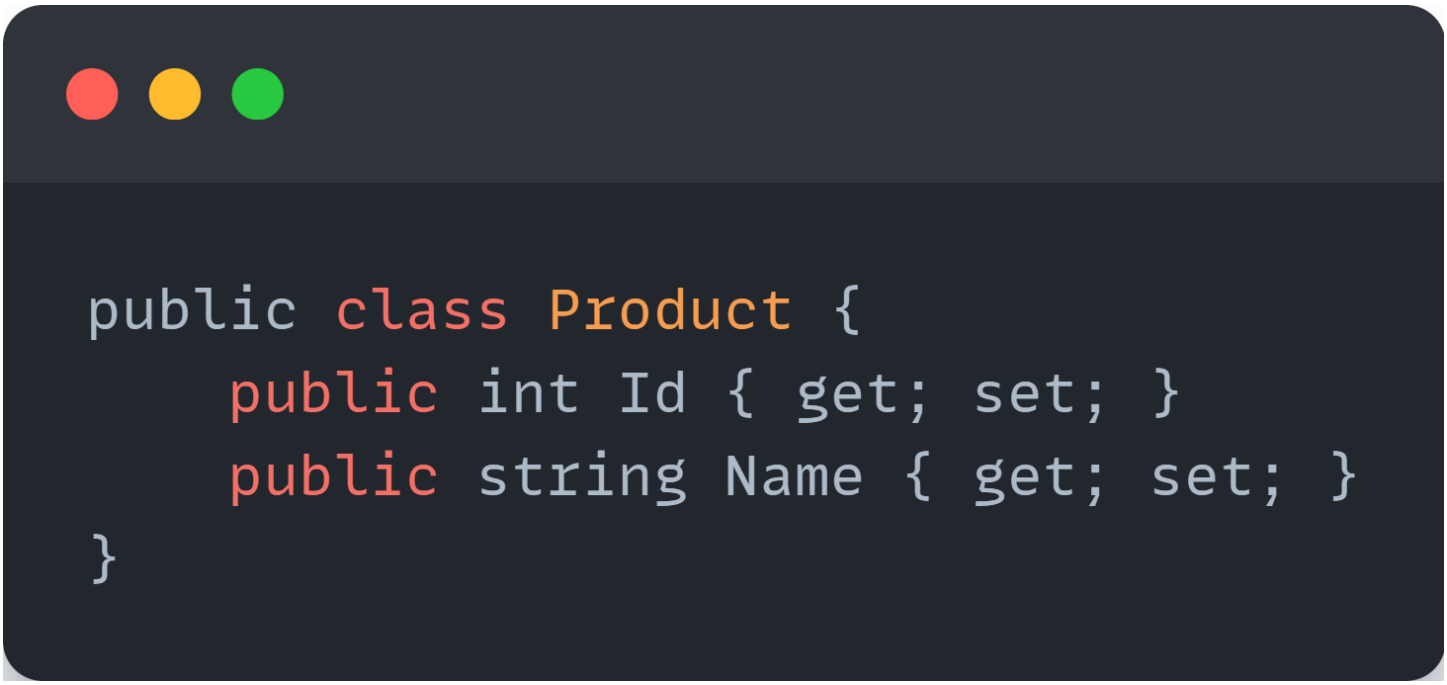


```
public IActionResult GetDetails(int id) {
    if (id ⩽ 0) return NotFound();
    return Json(new { Id = id, Name = "Product" });
}
```

51. What is Entity Framework Core?

Answer:
Entity Framework Core (EF Core) is an **ORM (Object-Relational Mapper)** for .NET that allows developers to interact with databases using C# objects instead of raw SQL queries.

Example:



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

With EF Core, you can perform database operations on **`Product`** without writing SQL.

52. What are the different approaches in EF Core?

Answer:
EF Core supports **three** main approaches:

1. **Code-First:** Define **C# classes** as models, and EF Core **creates the database**.
2. **Database-First:** Start with an **existing database** and **generate models** using EF tools.
3. **Hybrid:** A combination of Code-First and Database-First.

Example (Code-First Approach):

```
public class ApplicationDbContext : DbContext {
    public DbSet<Product> Products { get; set; }
}
```

53. How do you define relationships in EF Core?

Answer:
Relationships in EF Core are defined using **Navigation Properties** and **Fluent API**.

Example (One-to-Many Relationship):

```
public class Order {
    public int Id { get; set; }
    public List<Product> Products { get; set; }
}
public class Product {
    public int Id { get; set; }
    public int OrderId { get; set; }
    public Order Order { get; set; }
}
```

Fluent API Configuration:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>()
        .HasOne(p ⇒ p.Order)
        .WithMany(o ⇒ o.Products)
        .HasForeignKey(p ⇒ p.OrderId);
}
```

54. What is lazy loading and eager loading in EF Core?

Answer:

- **Lazy Loading**: Data is loaded **only when accessed**.
- **Eager Loading**: Data is **loaded immediately** with the main entity.

Example (Eager Loading):

```
var orders = _context.Orders.Include(o ⇒ o.Products).ToList();
```

This **loads orders with their products** in one query.

55. What are migrations in EF Core?

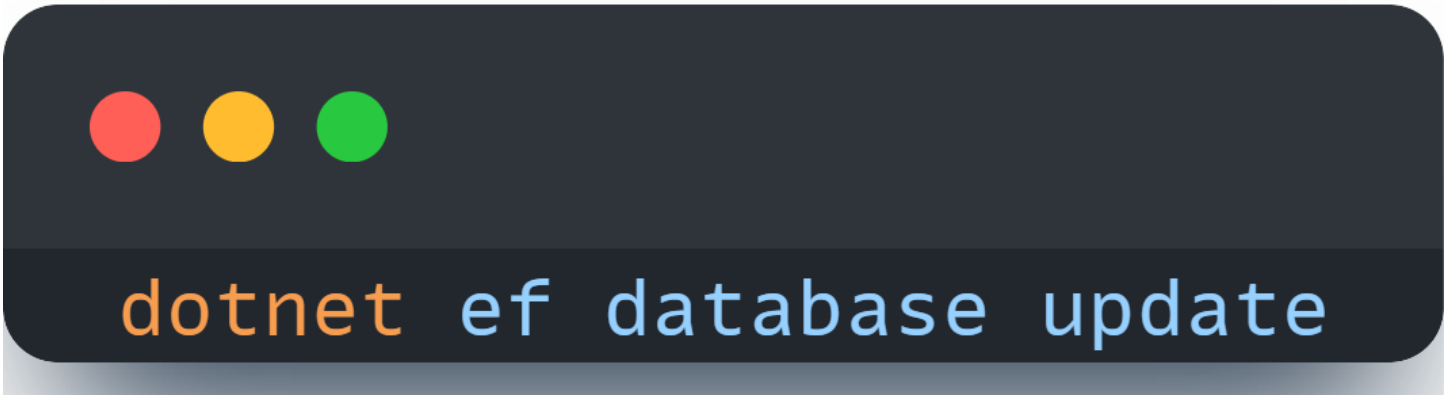
Answer:
Migrations allow **updating the database schema** when models change.

Steps to Use Migrations:

1. **Add Migration**

```
dotnet ef migrations add InitialCreate
```

2. Update Database



This applies schema changes without losing data.

56. What is the difference between DbContext and DbSet?

Answer:

Feature	DbContext	DbSet
Definition	Represents the **database connection**	Represents a **table in the database**
Purpose	Manages data operations	Works with entity records
Example	`ApplicationDbContext`	`DbSet<Product>`

Example:

```
public class ApplicationDbContext : DbContext {
    public DbSet<Product> Products { get; set; }
}
```

Here, Products is a **`DbSet`** that interacts with the "Products" table.

57. How do you perform CRUD operations in EF Core?

Answer:
CRUD stands for **Create, Read, Update, Delete**.

Create (Insert Data)

```
var product = new Product { Name = "Laptop" };
_context.Products.Add(product);
_context.SaveChanges();
```

Update Data

```
var product = _context.Products.Find(1);
product.Name = "Updated Laptop";
_context.SaveChanges();
```

Delete Data

```
var product = _context.Products.Find(1);
_context.Products.Remove(product);
_context.SaveChanges();
```

58. What is the purpose of the OnModelCreating method in EF Core?

Answer:
OnModelCreating is used to **configure model relationships, constraints, and database rules**.

Example: Custom Table Name

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Product>().ToTable("MyProducts");
}
```

This changes the table name from Products to **`MyProducts`.**

59. What is AsNoTracking in EF Core?

Answer:
AsNoTracking() **disables tracking** of retrieved entities, making queries **faster**.

Example:

```
var products = _context.Products.AsNoTracking().ToList();
```

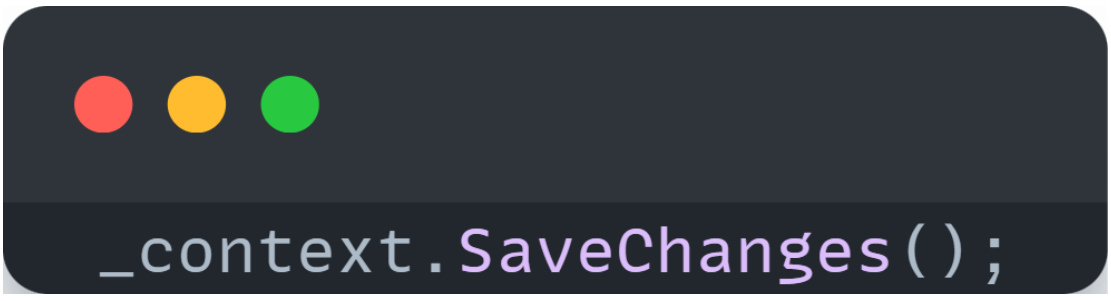
- **Without AsNoTracking()** → EF Core tracks changes.
- **With AsNoTracking()** → No tracking, **better performance** for read-only queries.

60. What is the difference between .SaveChanges() and .SaveChangesAsync()?

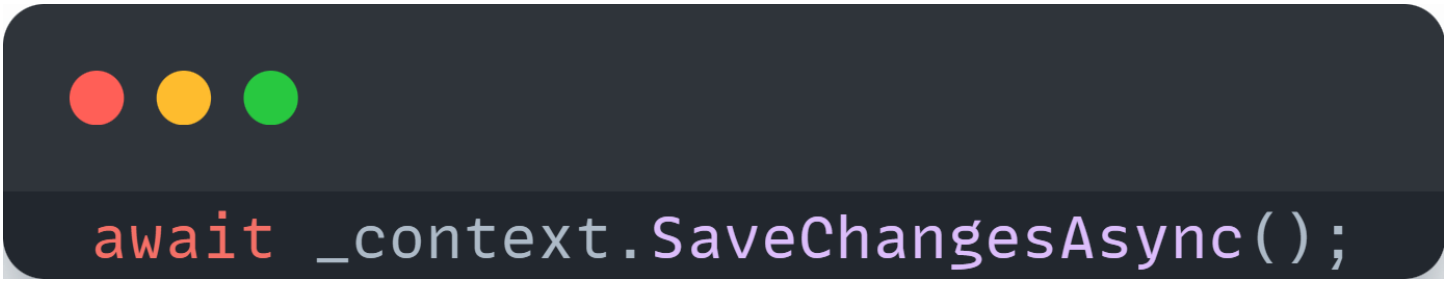
Answer:

Feature	`.SaveChanges()`	`.SaveChangesAsync()`
Type	**Synchronous**	**Asynchronous**
Performance	**Blocks execution** until done	**Non-blocking**, better for web apps
Use Case	Small apps, desktop apps	Web apps, APIs, large datasets

Example `(.SaveChanges() - Sync)`



Example (`.SaveChangesAsync() - Async`)



`SaveChangesAsync()` is **recommended** for better performance in **ASP.NET Core**.

61. What is a RESTful API?

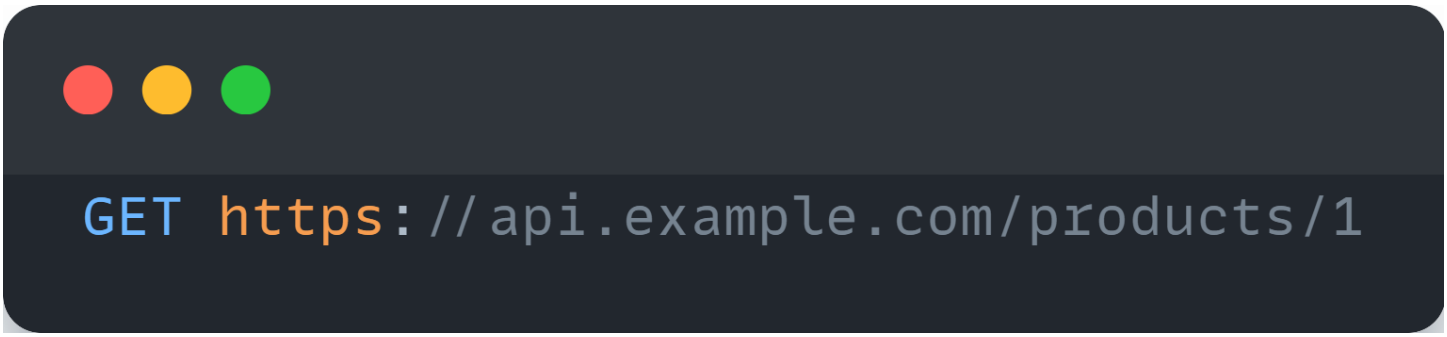
Answer:
A **RESTful API** (Representational State Transfer) is a **web service** that follows REST principles, allowing clients to communicate with a server using **HTTP methods**.

Key Features of RESTful APIs:

- Uses **stateless** communication
- Supports **JSON** and **XML**
- Follows **CRUD** (Create, Read, Update, Delete) operations

Example:

A RESTful API **GET request** to fetch a product:



This returns product details in **JSON** format.

62. What are the HTTP methods used in RESTful APIs?

Answer:
The main HTTP methods used in RESTful APIs are:

Method	Purpose	Example Endpoint
GET	Fetch data	`/products/1`
POST	Create new data	`/products`
PUT	Update existing data	`/products/1`
DELETE	Remove data	`/products/1`

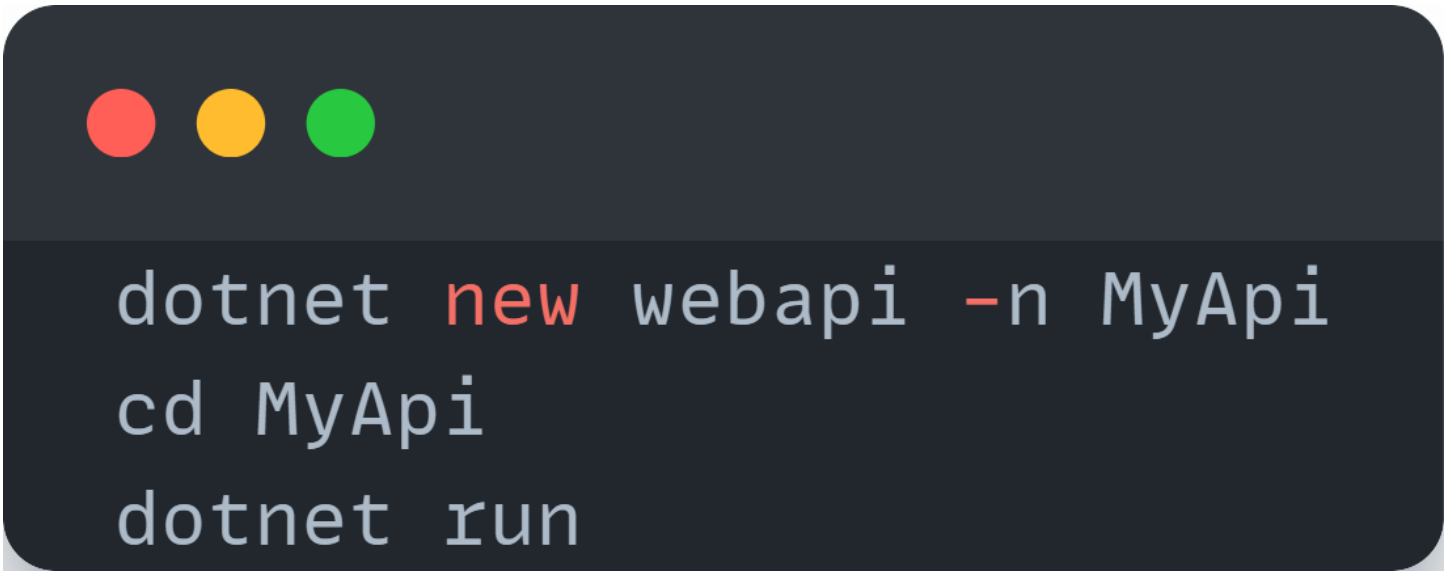
Example: (GET Request in C# using HttpClient)

```
HttpClient client = new HttpClient();
var response = await client.GetAsync("https://api.example.com/products/1");
var data = await response.Content.ReadAsStringAsync();
```

3. How do you create a Web API in .NET Core?

Answer:
To create a Web API in .NET Core:

1. **Create a new project**



2. **Define a Controller** (ProductsController.cs)

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase {
    [HttpGet]
    public IEnumerable<string> Get() {
        return new string[] { "Product1", "Product2" };
    }
}
```

3. **Run the API** and access http://localhost:5000/api/products.

64. What is attribute-based routing in ASP.NET Core Web API?

Answer:
Attribute-based routing allows defining API routes using **attributes** instead of defining them globally.

Example:

```
[Route("api/products")]
[ApiController]
public class ProductsController : ControllerBase {
    [HttpGet("{id}")]
    public IActionResult Get(int id) { return Ok($"Product {id}"); }
}
```

Here, GET `/api/products/1` will return "`Product 1`".

65. What is the difference between [HttpGet], [HttpPost], [HttpPut], and [HttpDelete]?

Answer:

Attribute	Purpose	Example Method
`[HttpGet]`	Fetch data	`Get(int id)`
`[HttpPost]`	Create data	`Post(Product product)`
`[HttpPut]`	Update data	`Put(int id, Product product)`
`[HttpDelete]`	Remove data	`Delete(int id)`

Example (`HttpPost` Method)

```
[HttpPost]
public IActionResult Create(Product product) {
    return CreatedAtAction(nameof(Get), new { id = product.Id }, product);
}
```

This adds a new product.

66. How do you implement authentication and authorization in ASP.NET Core Web API?

Answer:

1. **Use JWT Authentication**:
    - Install Microsoft.AspNetCore.Authentication.JwtBearer
    - Configure JWT in Program.cs

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.Authority = "https://your-auth-server.com";
        options.Audience = "your-api";
    });
```

    - Secure endpoints with [Authorize]

```
[Authorize] [HttpGet]
public IActionResult SecureData() { return Ok("This is a secured endpoint"); }
```

67. What are CORS and how do you enable them in Web API?

Answer:
CORS (Cross-Origin Resource Sharing) allows a server to handle requests from different origins (domains).

Enable CORS in .NET Core

1. Add CORS in Program.cs

```
builder.Services.AddCors(options ⇒ {
    options.AddPolicy("AllowAll", policy ⇒
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader());
});
```

2. Apply **CORS** policy

```
app.UseCors("AllowAll");
```

68. What is the difference between JSON and XML?

Answer:

Feature	JSON	XML
Format	Lightweight, uses `{}`	Uses `<tags>`
Readability	Easier to read	More complex
Data Size	Smaller	Larger
API Usage	Preferred for REST APIs	Used in older APIs

Example (JSON)

```
{
  "name": "Laptop", "price": 1000
}
```

Example (XML)

```
<Product>
  <Name>Laptop</Name> <Price>1000</Price>
</Product>
```

69. What is Swagger in ASP.NET Core?

Answer:
Swagger is a tool to document and test Web APIs interactively.

How to Enable Swagger in .NET Core

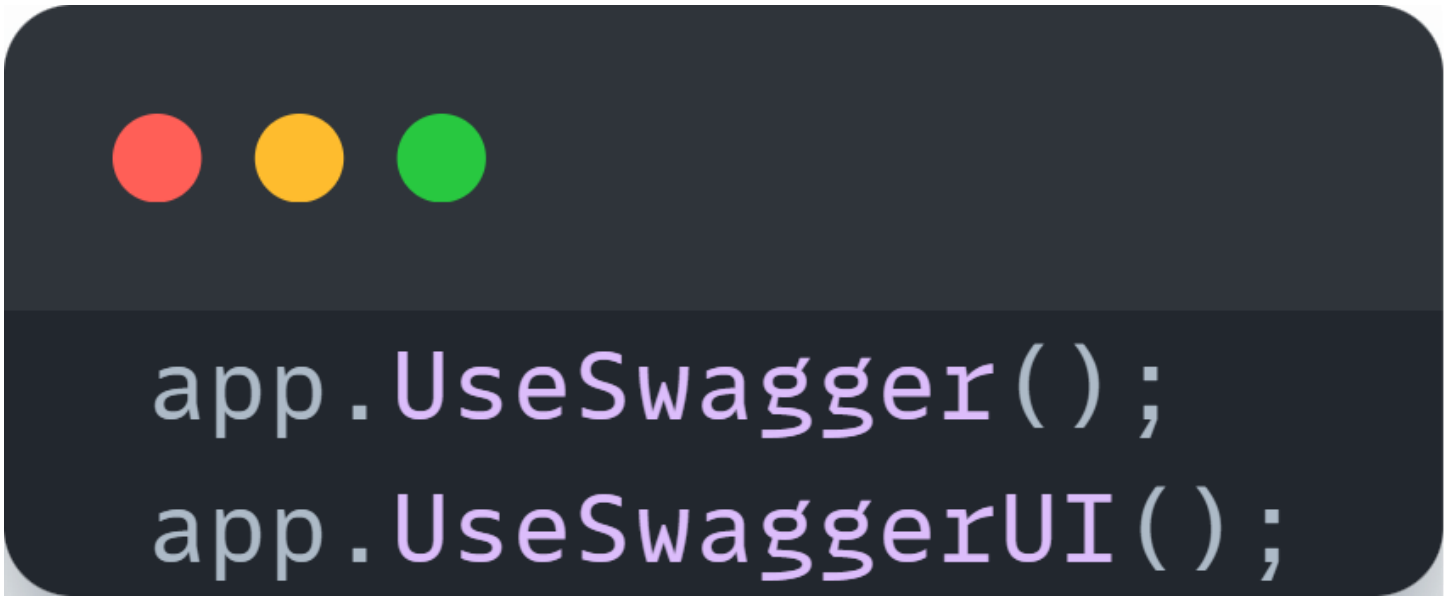
1. **Install NuGet Package**

```
dotnet add package Swashbuckle.AspNetCore
```

2. **Configure in Program.cs**

```
builder.Services.AddSwaggerGen();
```

3. **Enable in Middleware**



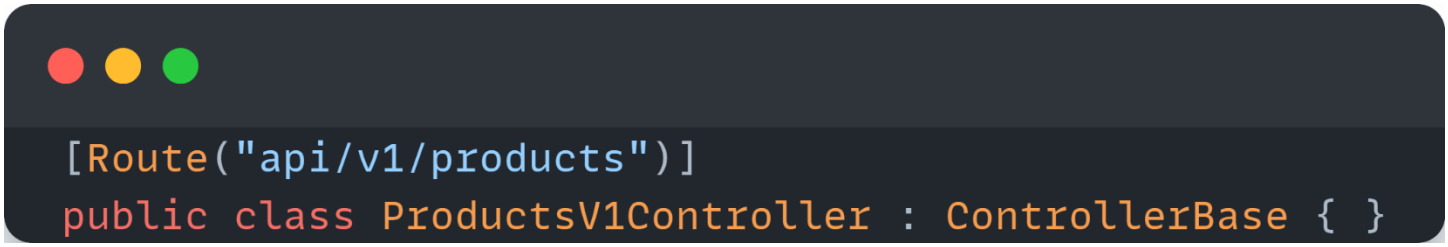
Now, access Swagger UI at http://localhost:5000/swagger.

70. What is versioning in Web API?

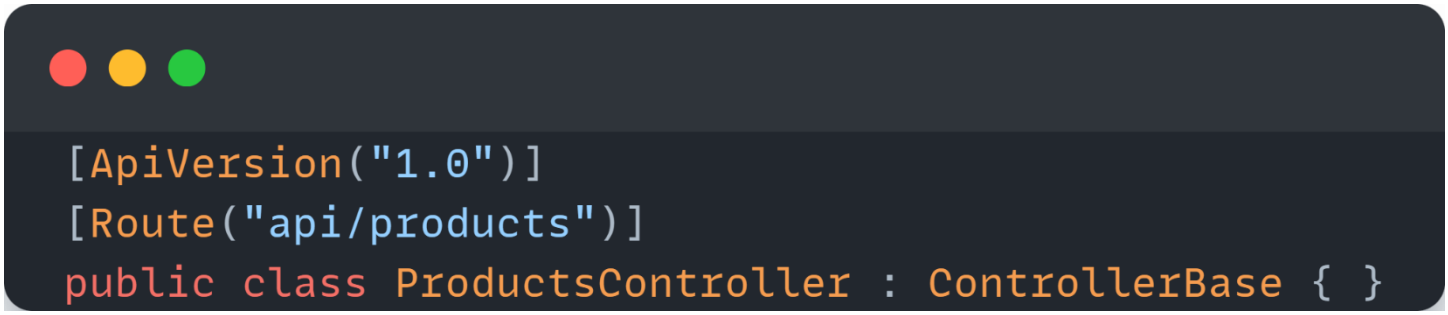
Answer:
API versioning allows multiple versions of an API to exist simultaneously.

Methods to Implement API Versioning:

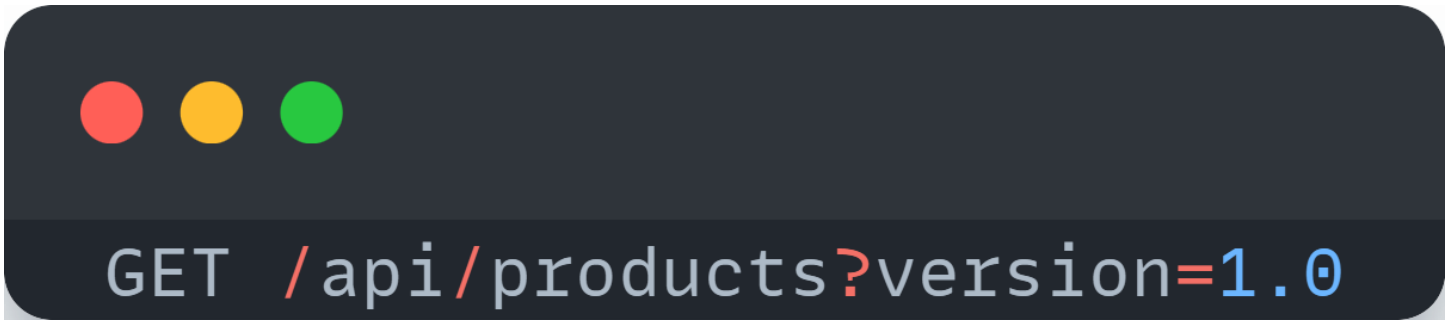
1. **URL-based versioning**



2. **Header-based versioning**



3. **Query string versioning**



71. What is a primary key and a foreign key?

Answer:

Primary Key:

A primary key is a unique identifier for a record in a table.

- Cannot be NULL
- Must be unique

Foreign Key:

A foreign key establishes a relationship between two tables by referencing the primary key of another table.

Example:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(50)
);
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Here, **CustomerID** in Orders is a foreign key referring to **`Customers.CustomerID`**.

72. What is normalization in databases?

Answer:

Normalization is the process of organizing data in a database to **eliminate redundancy** and improve data integrity.

Normalization Forms:

Form	Purpose
1NF	Remove duplicate columns & ensure atomicity
2NF	Remove partial dependencies
3NF	Remove transitive dependencies
BCNF	Advanced version of 3NF

73. Explain the different types of SQL Joins.

Answer:

Types of Joins:

Join Type	Description
INNER JOIN	Returns matching records from both tables
LEFT JOIN	Returns all records from the left table + matches from the right
RIGHT JOIN	Returns all records from the right table + matches from the left
FULL JOIN	Returns all records when there is a match in either table

Example (`INNER JOIN`)

```sql
SELECT Customers.Name, Orders.OrderID
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This retrieves customers who have placed **orders**.

74. What is an index in SQL?

Answer:

An **index** improves the speed of data retrieval in a table.

Types of Indexes:

- **Clustered Index:** Sorts data physically in table
- **Non-clustered Index:** Stores pointers to data instead of sorting

Example (Creating an Index)

```sql
CREATE INDEX idx_customer_name ON Customers(Name);
```

This improves **search performance** for Name.

75. What is the difference between INNER JOIN and OUTER JOIN?

Answer:

Join Type	Description
INNER JOIN	Returns only matching rows from both tables
OUTER JOIN	Returns all rows from one or both tables, even if there's no match

Example (`LEFT JOIN - Outer Join`)

```sql
SELECT Customers.Name, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This returns all customers, even those who haven't placed orders.

76. What is a stored procedure?

Answer:

A **stored procedure** is a precompiled SQL script that can be executed multiple times.

Example:

```sql
CREATE PROCEDURE GetOrdersByCustomer
    @CustomerID INT
AS
BEGIN
    SELECT * FROM Orders WHERE CustomerID = @CustomerID;
END;
```

Run the stored procedure:

```sql
EXEC GetOrdersByCustomer @CustomerID = 1;
```

77. What is the difference between DELETE and TRUNCATE?

Answer:

Operation	DELETE	TRUNCATE
Removes Data?	Yes, based on condition	Yes, entire table
Can be Rolled Back?	Yes, if in a transaction	No
Performance	Slower	Faster
Resets Identity Column?	No	Yes

Example (`DELETE` vs `TRUNCATE`)

```sql
DELETE FROM Orders WHERE OrderID = 1;  -- Removes specific row
TRUNCATE TABLE Orders;  -- Removes all rows
```

78. How do you use transactions in SQL?

Answer:

A transaction ensures multiple SQL operations are executed as a single unit.

Example (`Using a Transaction`)

```sql
BEGIN TRANSACTION;
UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
-- If everything is fine, commit
COMMIT;
-- If there's an error, rollback
ROLLBACK;
```

79. What is the purpose of the GROUP BY clause?

Answer:

`GROUP BY` is used to group rows with the same values and apply aggregate functions.

Example:

```sql
SELECT CustomerID, COUNT(OrderID) AS OrderCount
FROM Orders
GROUP BY CustomerID;
```

This returns the number of orders per customer.

80. What is the difference between HAVING and WHERE?

Answer:

Clause	Purpose	Works With
WHERE	Filters rows **before** grouping	Regular conditions
HAVING	Filters groups **after** aggregation	Aggregate functions

Example:

```sql
SELECT CustomerID, COUNT(OrderID) AS OrderCount
FROM Orders
GROUP BY CustomerID
HAVING COUNT(OrderID) > 5;
```

This returns customers who placed more than 5 orders.

81. What are authentication and authorization?

Answer:

Authentication:

- The process of verifying a user's **identity** (Who are you?).
- Example: Logging in with a **username and password**.

Authorization:

- The process of verifying a user's **permissions** (What can you access?).
- Example: A **user** can view data, but only an **admin** can delete it.

Example in .NET Core

```csharp
[Authorize] // Requires authentication
public IActionResult SecurePage()
{
    return View();
}
```

This ensures that only authenticated users can access **`SecurePage()`.**

82. What are JWT tokens, and how are they used in .NET Core?

Answer:

JWT (JSON Web Token)

- A secure, compact way to **transmit user authentication data** between parties.
- Used in **stateless authentication**, meaning no session is stored on the server.

Structure of a JWT:

1. **Header** – Algorithm & Token Type
2. **Payload** – User data (claims)
3. **Signature** – Verifies token authenticity

Example (Generating a JWT Token in .NET Core)

```
var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("your_secret_key"));
var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
var token = new JwtSecurityToken(
    issuer: "yourdomain.com",
    audience: "yourdomain.com",
    expires: DateTime.Now.AddHours(1),
    signingCredentials: credentials
);
var jwt = new JwtSecurityTokenHandler().WriteToken(token);
```

This generates a **JWT** token that expires in **1 hour**.

83. What is OAuth 2.0?

Answer:

OAuth 2.0 is an **authorization framework** that allows third-party applications to access user data without exposing passwords.

How OAuth 2.0 Works?

1. User logs in and **authorizes** the application.
2. The application requests an **access token** from the OAuth provider.
3. The application uses the token to **access user data**.

Example OAuth 2.0 Providers

- Google Sign-In
- Facebook Login
- GitHub OAuth

OAuth 2.0 in .NET Core

```
services.AddAuthentication()
    .AddGoogle(options ⇒
    {
        options.ClientId = "your-client-id";
        options.ClientSecret = "your-client-secret";
    });
```

This **enables Google authentication** in an ASP.NET Core application.

84. What is Identity Server in .NET Core?

Answer:

Identity Server is an open-source authentication server for handling OAuth 2.0 and OpenID Connect.

Why Use Identity Server?

✓ Centralized authentication for multiple apps
✓ Supports JWT, OAuth 2.0, and OpenID Connect
✓ Enables Single Sign-On (SSO)

Example: Adding Identity Server to .NET Core

```
services.AddIdentityServer()
    .AddDeveloperSigningCredential() // For development
    .AddInMemoryApiScopes(new List<ApiScope> { new ApiScope("api1", "My API") });
```

This sets up **Identity Server** with **basic API security**.

85. How do you implement role-based authorization in .NET Core?

Answer:

Role-based authorization allows **different users** to have **different access levels** based on their roles.

Example: Assigning Roles

```
[Authorize(Roles = "Admin")]
public IActionResult AdminPanel()
{
    return View();
}
```

Only **users with the 'Admin' role** can access `AdminPanel()`.

Defining Roles in Program.cs:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdminRole", policy => policy.RequireRole("Admin"));
});
```

Checking Roles in Code:

```
if (User.IsInRole("Admin"))
{
    // Grant access
}
```

This checks the user's **role** before allowing actions.

86. What is unit testing?

Answer:

Unit testing is a software testing technique where individual **components** (methods, functions, classes) are tested **independently** to ensure they work as expected.

Why Use Unit Testing?

✓ Catches bugs early in development
✓ Ensures code reliability and maintainability
✓ Helps with refactoring without breaking existing functionality

Example: Unit Test for a Calculator Class

```
public class Calculator
{
    public int Add(int a, int b) ⇒ a + b;
}
```

Test Case (Using xUnit)

```
using Xunit;

public class CalculatorTests
{
    [Fact]
    public void Add_ShouldReturnSum()
    {
        var calc = new Calculator();
        var result = calc.Add(3, 2);
        Assert.Equal(5, result); // Check if the sum is correct
    }
}
```

87. What are xUnit, NUnit, and MSTest?

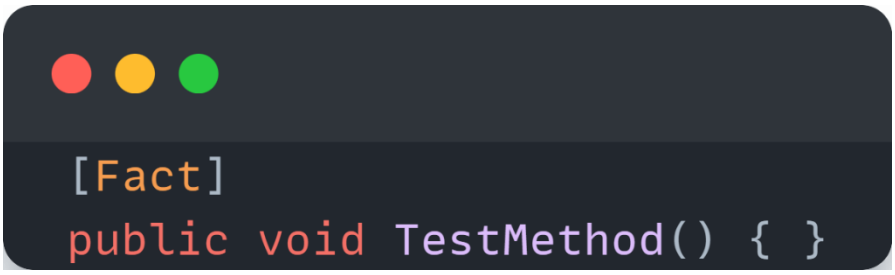
Answer:

These are popular **unit testing frameworks** for .NET.

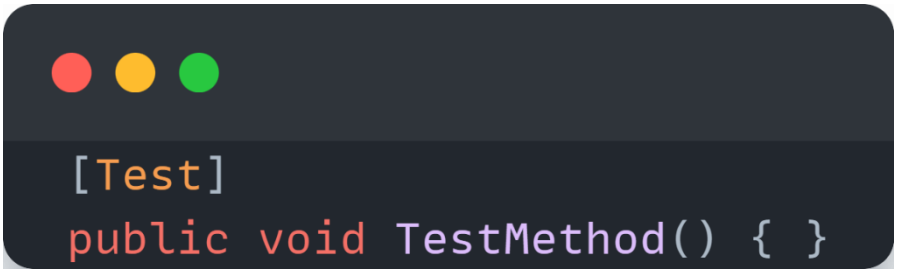
Framework	Features
xUnit	Modern, supports dependency injection, used in .NET Core
NUnit	Popular, supports attributes like `[Test]`, `[SetUp]`
MSTest	Built-in framework from Microsoft, used in older projects

Example: xUnit vs. NUnit vs. MSTest

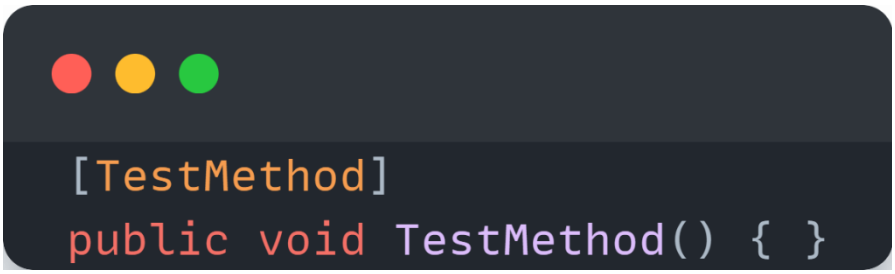
xUnit Test



NUnit Test



MSTest Test



88. What is mocking in unit testing?

Answer:

Mocking is a technique used to simulate dependencies in unit tests. It helps test a class without needing the actual database, API, or external service.

Why Use Mocking?

✓ Removes external dependencies (e.g., databases, APIs)

✓ Improves test speed

✓ Allows testing edge cases easily

Example: Mocking with Moq

Let's assume we have a service that depends on a database:

```
public interface IUserRepository
{
    string GetUserName(int id);
}
```

We mock the repository instead of calling a real database:

```
using Moq;
using Xunit;
public class UserServiceTests
{
    [Fact]
    public void GetUserName_ShouldReturnMockedName()
    {
        var mockRepo = new Mock<IUserRepository>();
        mockRepo.Setup(repo => repo.GetUserName(1)).Returns("Afzal");
        var result = mockRepo.Object.GetUserName(1);
        Assert.Equal("Afzal", result);
    }
}
```

89. How do you write a simple unit test in .NET Core?

Answer:

Steps to Write a Unit Test in .NET Core

1. Create a **Test Project** (using xUnit, NUnit, or MSTest).
2. Write **Test Methods** using Assert statements.
3. Run tests using **Test Explorer** in Visual Studio or CLI.

Example: Testing a String Utility Class

```
public class StringHelper
{
    public bool IsPalindrome(string input) => input == new string(input.Reverse().ToArray());
}
```

Unit Test (Using xUnit):

```
using Xunit;
public class StringHelperTests
{
    [Fact]
    public void IsPalindrome_ShouldReturnTrueForPalindrome()
    {
        var helper = new StringHelper();
        var result = helper.IsPalindrome("madam");
        Assert.True(result);
    }
}
```

90. How do you debug an ASP.NET Core application?

Answer:

Techniques for Debugging in ASP.NET Core

✓ **Using Breakpoints** – Pause execution at specific lines
✓ **Debugging with Visual Studio** – Step through code line by line

✓ **Logging (Serilog, NLog)** – Capture application errors
✓ **Using Debug.WriteLine() or Console.WriteLine()** – Print debug messages
✓ **Attaching Debugger to Running Process** – Debug a deployed app

91. How do you optimize the performance of an ASP.NET Core application?

Answer:

Optimizing performance ensures a **fast and scalable** application. Here are key strategies:

✓ **Use Response Caching** – Store frequently used responses
✓ **Enable Compression** – Reduce response size using Gzip
✓ **Use Asynchronous Code (async/await)** – Avoid blocking calls
✓ **Optimize Database Queries** – Avoid unnecessary queries and use indexing
✓ **Use Load Balancing** – Distribute traffic among multiple servers
✓ **Use Profiling Tools** – Identify bottlenecks with **Application Insights**

Example: Enable Response Compression in ASP.NET Core

Add this in **`Program.cs`**:

```
builder.Services.AddResponseCompression();
```

In **`Middleware`**:

```
app.UseResponseCompression();
```

Benefit: Reduces response size, improving speed.

92. What are caching techniques in .NET Core?

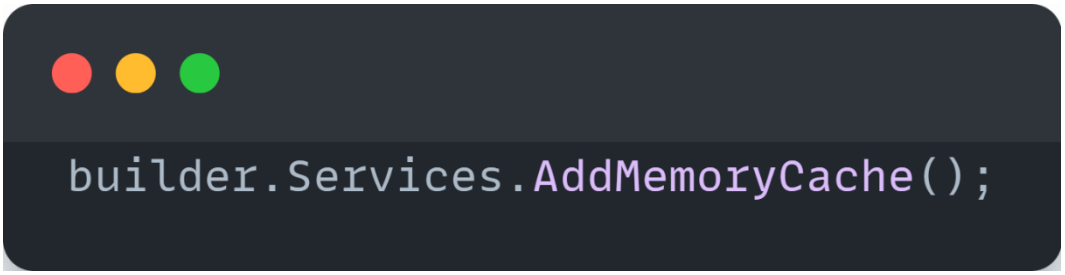
Answer:

Caching reduces load times by storing frequently used data. There are 3 main caching types:

Caching Type	Description	Example
In-Memory Caching	Stores data in server memory	`MemoryCache`
Distributed Caching	Stores data across multiple servers	Redis, SQL Server cache
Response Caching	Stores HTTP responses	`ResponseCache` attribute

Example: In-Memory Caching in ASP.NET Core

Add caching in **`Program.cs`**:



```
builder.Services.AddMemoryCache();
```

Use caching in the controller:

```
public class ProductController : Controller
{
    private readonly IMemoryCache _cache;
    public ProductController(IMemoryCache cache)
    {
        _cache = cache;
    }
    public IActionResult GetProducts()
    {
        if (!_cache.TryGetValue("products", out List<string> products))
        {
            products = new List<string> { "Laptop", "Phone", "Tablet" };
            _cache.Set("products", products, TimeSpan.FromMinutes(10));
        }
        return Ok(products);
    }
}
```

Benefit: Faster response time by avoiding repetitive database calls.

93. How do you improve the performance of Entity Framework Core queries?

Entity Framework (EF) Core can slow down performance if not optimized. Here are best practices:

✓ **Use AsNoTracking()** – Improves read-only query performance
✓ **Avoid Lazy Loading** – Use **`Eager Loading (Include())`** instead
✓ **Use Select() for Projection** – Fetch only required columns
✓ **Batch Queries** – Reduce the number of database calls
✓ **Optimize Indexes** – Ensure database indexes exist for frequent queries

Example: Using **`AsNoTracking()`** for Read-Only Queries

```
var users = _context.Users.AsNoTracking().ToList();
```

Benefit: Increases performance by disabling entity tracking.

94. What are asynchronous programming best practices in .NET Core?

Answer:

Using async/await properly improves performance and responsiveness.

✓ **Use async for I/O operations –** (e.g., API calls, database queries)
✓ **Avoid Task.Wait() or .Result –** Blocks the thread
✓ **Use ConfigureAwait(false) –** For background tasks
✓ **Return Task instead of void –** Avoids unhandled exceptions
✓ **Use CancellationToken –** Allows task cancellation

Example: Async Database Query in EF Core

```
public async Task<List<User>> GetUsersAsync()
{
    return await _context.Users.ToListAsync();
}
```

Benefit: Avoids blocking and improves scalability.

95. How do you handle memory leaks in C#?

A memory leak happens when memory is allocated but not released, causing performance issues.

✓ **Use IDisposable and using Statement –** Free resources properly
✓ **Avoid Static References to Large Objects –** Causes unnecessary memory retention
✓ **Use Weak References –** Helps garbage collection
✓ **Monitor Memory Usage –** Use tools like dotMemory, PerfMon, and GC.Collect()

Example: Properly Disposing Objects in C#

```
public void ReadFile()
{
    using (StreamReader reader = new StreamReader("file.txt"))
    {
        string content = reader.ReadToEnd();
    } // FileStream is automatically closed here
}
```

Benefit: Prevents file handle leaks by releasing memory after use.

96. What are .NET Core CLI commands?

.NET Core CLI (Command Line Interface) is a tool for building, running, and managing .NET applications without using Visual Studio.

Common .NET Core CLI Commands:

Command	Description
`dotnet new console -o MyApp`	Creates a new Console App in a folder named **MyApp**
`dotnet new mvc -o WebApp`	Creates a new MVC project named **WebApp**
`dotnet build`	Builds the project
`dotnet run`	Runs the application
`dotnet test`	Runs unit tests
`dotnet publish -c Release -o publish`	Publishes the project
`dotnet ef migrations add InitialCreate`	Adds a new EF Core migration
`dotnet ef database update`	Updates the database schema

97. What is Docker, and how is it used in .NET Core?

Docker is a containerization tool that allows you to package your .NET Core app with all dependencies into a lightweight, portable container.

How .NET Core Uses Docker

1. Write a Dockerfile (Defines how to build the container)
2. Build the Docker Image
3. Run the Container

Example: Dockerfile for a .NET Core App

```
# Use official .NET Core runtime image
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY . .
ENTRYPOINT ["dotnet", "MyApp.dll"]
```

Commands to Run Docker Container:

```
docker build -t myapp .
docker run -d -p 8080:80 myapp
```

98. What is SignalR in .NET Core?

SignalR is a real-time communication library for WebSockets-based applications.

Use Cases of SignalR

✓ Live Chat Applications
✓ Live Notifications (e.g., Facebook, Twitter updates)
✓ Live Dashboards & Stock Market Apps

How SignalR Works in .NET Core

1. nstall SignalR Package

```
dotnet add package Microsoft.AspNetCore.SignalR
```

2. Create a SignalR Hub

```
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

3. Configure SignalR in Startup

```
app.UseEndpoints(endpoints ⇒
{
    endpoints.MapHub<ChatHub>("/chatHub");
});
```

99. What is gRPC in .NET Core?

gRPC is a high-performance, RPC framework used for communication between microservices.

gRPC vs REST API

Feature	gRPC	REST API
Protocol	Uses HTTP/2	Uses HTTP/1.1
Data Format	Binary (Protobuf)	Text (JSON/XML)
Speed	🚀 Faster	🐢 Slower
Use Case	Microservices & internal APIs	Public APIs

Example: Defining a gRPC Service

1. Define a Proto File (greet.proto)

```
syntax = "proto3";
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
```

2. Generate C# Code from Proto File

```
dotnet grpc --proto greet.proto
```

3. mplement the gRPC Service

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = $"Hello, {request.Name}!"
        });
    }
}
```

100. What is the latest version of .NET, and what are its new features?

Answer:

The latest version of .NET is indeed .NET 9, released on November 12, 2024.

Key Features:

- **Performance Improvements –** Over 1,000 optimizations in runtime and workloads.
- **AI Integration –** New TensorPrimitives and Tensor<T> for machine learning**.**
- **Minimal API Enhancements –** Less boilerplate, faster development.
- **Improved LINQ Methods –** New methods like CountBy, AggregateBy.
- **Built-in OpenAPI Support –** Easier API documentation in ASP.NET Core.
- **Better Garbage Collection –** Reduced memory usage, improved efficiency.