Asp.Net Core

What .NET Represents?

NET stands for Network Enabled Technology. In .NET, dot (.) refers to object-oriented and NET refers to the internet. So the complete .NET means through object-oriented we can implement internet-based applications

What is a Framework?

A Framework is a Software. Or you can say a framework is a collection of many small technologies integrated together to develop applications that can be executed anywhere.

What is .NET Framework?

According to Microsoft, .NET Framework is a software development framework for building and running applications on Windows. The .NET Framework is part of the .NET platform, a collection of technologies for building apps for Linux, macOS, Windows, iOS, Android, and more.

Different Types of .NET Framework.

.NET Framework vs .NET Core

Platform:.NET Framework is designed specifically for Windows-based applications, whereas .NET Core is cross-platform, allowing development on Windows, macOS, and Linux.

Performance:.NET Framework has more dependencies, making it heavier and slightly slower in performance. .NET Core is optimized for high performance and scalability.

Deployment:Applications built with .NET Framework require a full installation on the system. In contrast, .NET Core applications can be deployed as self-contained applications without requiring installation on the target system.

Microservices Support:.NET Framework offers limited support for microservices and container-based applications. .NET Core, on the other hand, is specifically designed for microservices and has full support for Docker and Kubernetes.

Application Types:.NET Framework is best suited for Windows Forms, WPF, ASP.NET Web Forms, and legacy applications. .NET Core is ideal for modern cloud applications, web APIs, and microservices.

Development Model:.NET Framework follows a monolithic runtime model, whereas .NET Core is modular and lightweight, allowing multiple runtime versions to coexist.

Open-Source:.NET Framework is only partially open-source, whereas .NET Core is fully open-source and community-driven.

Future Support:.NET Framework is not receiving major updates and is transitioning towards .NET 6 and later versions. .NET Core (now unified under .NET 6+) is actively developed and represents the future of .NET.

Versioning & Updates:.NET Framework maintains fixed version updates with a focus on backward compatibility. .NET Core follows a more dynamic update cycle, with frequent updates that may introduce breaking changes when necessary.

Xamarin/Mono is a .NET implementation for running apps on all the major mobile operating systems, including iOS and Android.

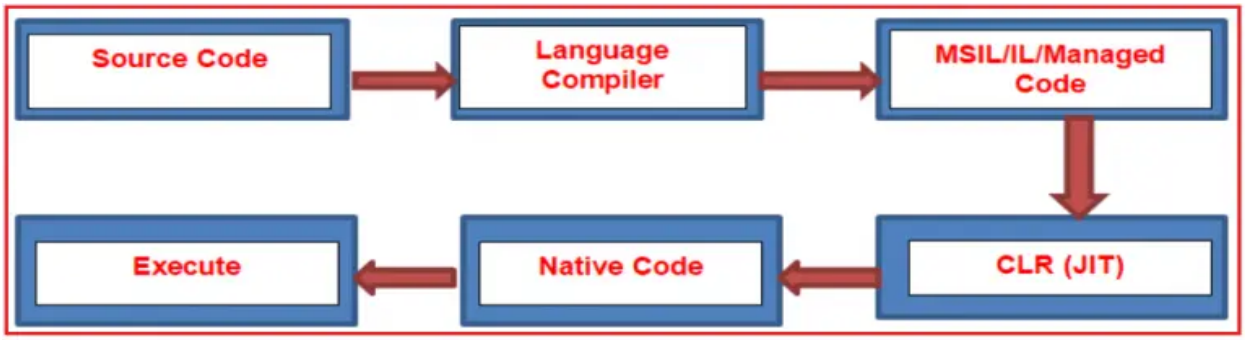
What does the .NET Framework Provide?

CL (Class Libraries) -helps to write the program and run the program

CLR (Common Language Runtime)

Common Language Runtime (CLR): helps to compile the program or execute

CLR stands for Common Language Runtime and it is the core component under the .NET framework which is responsible for converting the MSIL (Microsoft Intermediate Language) code into native code and then provides the runtime environment to execute the code. That means Common Language Runtime (CLR) is the execution engine that handles running applications. It provides services like thread management, garbage collection, type safety, exception handling, and more. In our next article, we will discuss **CLR** in detail. System.GC.Collect()



What is JIT?

JIT stands for the **Just-in-Time** compiler. It is the component of **CLR** that is responsible for converting **MSIL** code into **Native Code**. Native code is the code that is directly understandable by the operating system.

Garbage Collector:

When a dot net application runs, lots of objects are created. At a given point in time, it is possible that some of those objects are not used by the application. So, **Garbage Collector in .NET Framework** is nothing but a **Small Routine** or you can say it's a **Background Process Thread** that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

Managed and Unmanaged Code in .NET

1. Managed Code

Managed code is code that runs under the supervision of the Common Language Runtime (CLR) in the .NET framework. The CLR provides several benefits, including automatic memory management (Garbage Collection), type safety, exception handling, and security.

Characteristics of Managed Code:

- Runs inside the CLR (Common Language Runtime).
- Garbage Collection (GC) automatically handles memory management.
- Provides security features such as Code Access Security (CAS).

- Supports Just-In-Time (JIT) Compilation, converting Intermediate Language (IL) into native code.
- Platform-independent since it relies on CLR for execution.

2. Unmanaged Code

Unmanaged code is any code that is **executed directly by the operating system**, outside of the CLR's control. It is typically written in **C, C++, or assembly language** and does not benefit from CLR features like garbage collection or runtime security.

Characteristics of Unmanaged Code:

- **Does not run under CLR**; instead, it interacts directly with the OS.
- Developers must **manually manage memory allocation and deallocation**.
- Prone to **memory leaks and security vulnerabilities** if not handled properly.
- Provides **better performance** in scenarios where direct memory manipulation is required.

Conclusion

- Use Managed Code for most .NET applications since it provides security, memory management, and ease of development.
- Use Unmanaged Code when interacting with low-level system resources, external libraries, or performance-critical applications.

Understanding .DLL and .EXE Files in .NET

1. What is a .DLL File?

DLL (Dynamic Link Library) files are reusable libraries that contain code and data that can be used by multiple programs. They do not have a main entry point and cannot run independently.

Key Characteristics of .DLL Files

- Used for Code Reusability – Allows sharing of common functionality across different applications.
- Cannot Be Executed Independently – A .DLL file needs to be referenced by an EXE or another assembly.
- Encapsulation of Logic – Helps in modularizing applications by separating concerns.
- Supports Multiple Applications – Different applications can use the same .DLL file, reducing code duplication.
- Example Usage: Common utility functions, third-party libraries, and UI components.

2. What is an .EXE File?

EXE (Executable) files are applications that contain a **Main() method** and can be executed independently.

Key Characteristics of .EXE Files

- **Contains an Entry Point** – The `Main()` method defines where execution starts.
- **Standalone Execution** – Can run without needing another application.
- **Typically Used for Applications** – Console applications, Windows applications, and services.
- **May Depend on DLLs** – EXE files often reference DLLs for additional functionality.

5. Conclusion

- **DLL (.dll) files** are libraries that **cannot execute on their own** but provide reusable functionality to other applications.
- **EXE (.exe) files** are standalone applications that **contain an entry point (`Main()`) and can be executed independently**.
- **EXE files often depend on DLLs** to avoid code duplication and improve modularity.
- **.NET applications use both EXE and DLL files** to build scalable and maintainable solutions.

What is Middleware?

Middleware is a software component that sits between the web server and the application. It intercepts incoming requests and outgoing responses and allows you to modify them. Middleware can be used to perform a wide range of tasks such as authentication, logging, compression, and caching.

Types of Middleware

Terminal Middleware

The terminal middleware is responsible for sending the response back to the client. It is the final middleware component in the pipeline. Terminal middleware can be used to modify the outgoing response before it is sent back to the client.

Some examples of terminal middleware include:

Static files middleware: This middleware is used to serve static files such as CSS, JavaScript, and images.

- File server middleware: This middleware is used to serve files from a specified directory.
- MVC middleware: This middleware is used to handle requests for MVC endpoints.

Non-terminal Middleware

Non-terminal middleware is any middleware component that is not the final component in the pipeline. Non-terminal middleware can be used to modify incoming requests and outgoing responses.

Some examples of non-terminal middleware include:

Authentication middleware: This middleware is used to authenticate users.

Authorization middleware: This middleware is used to authorize users to access certain resources.

Response compression middleware: This middleware is used to compress the response before it is sent back to the client.

Request logging middleware: This middleware is used to log requests.

Routing middleware: This middleware is used to route requests to the appropriate endpoint.

Custom Middleware

In addition to the built-in middleware components that are available in .NET Core, you can also create your own custom middleware components. Creating custom middleware allows you to add functionality to your application that is specific to your needs.

To create custom middleware, you need to create a class that implements the IMiddleware interface. The IMiddleware interface has a single method called InvokeAsync that is called when the middleware component is invoked.

```
public class CustomHeaderMiddleware : IMiddleware

{

   public async Task InvokeAsync(HttpContext context, Func<Task> next)

    {

        context.Response.Headers.Add("X-Custom-Header", "Hello World");

        await next();

    }

}
```

In this example, the CustomHeaderMiddleware class implements the IMiddleware interface. The InvokeAsync method takes two parameters: the HttpContext and a Func<Task> delegate that represents the next middleware component in the pipeline.

In this example, the middleware adds a custom header called “X-Custom-Header” to the outgoing response. It then calls the next middleware component in the pipeline by invoking the next delegate.

To use the custom middleware component, you need to add it to the middleware pipeline in your application’s Startup class. You can do this by calling the UseMiddleware method on the IApplicationBuilder instance.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

  app.UseMiddleware<CustomHeaderMiddleware>();

  app.UseRouting();

  app.UseEndpoints(endpoints =>

  {

     endpoints.MapControllers();

  });

}
```

Best Practices for Using Middleware

Here are some best practices for using middleware in your .NET Core applications:

1. Use middleware sparingly: Middleware can be a powerful tool, but it should be used sparingly. Adding too much middleware to the pipeline can slow down your application and make it more difficult to maintain.
2. Order middleware components carefully: The order in which middleware components are added to the pipeline is important. Make sure that you add the middleware components in the correct order so that they are executed in the desired sequence.
3. Use terminal middleware components when appropriate: When you have a middleware component that is the final component in the pipeline, use terminal middleware. This ensures that the response is sent back to the client and that no further middleware components are executed.
4. Test your middleware: Before deploying your application, make sure that you test your middleware components thoroughly. This will help you identify any issues or performance problems.

Is There Any Class to Inherit in a Middleware Class in .NET Core?

No, there is no specific base class that a middleware class must inherit in .NET Core. Middleware in ASP.NET Core is simply a class with a `RequestDelegate` parameter and an `Invoke` method. However, middleware follows a convention rather than an inheritance model.

Q: "Do we need to inherit any class to create middleware in .NET Core?"

A: "No, middleware in .NET Core does not require inheriting a base class. It only needs an `Invoke` method. However, we can implement the `IMiddleware` interface if we want built-in dependency injection support."

✅ You **don't need to inherit** any class for middleware in .NET Core.

✅ You can use **standard middleware** or **implement `IMiddleware`** for DI support.

✅ Middleware is registered using `app.UseMiddleware<T>()` in `Program.cs`.

Dependency Injection (DI) in .NET Core

1. Introduction

Dependency Injection (DI) is a design pattern and a built-in feature of ASP.NET Core that facilitates the creation and management of dependencies in a loosely coupled manner. It enhances code maintainability, testability, and reusability by injecting dependencies rather than instantiating them directly within a class.

2. Benefits of Dependency Injection

- Loose Coupling: Reduces dependencies between classes, making them more modular.
- Testability: Easier unit testing as dependencies can be replaced with mocks.
- Maintainability: Improved code structure and separation of concerns.
- Automatic Dependency Management: .NET Core's built-in DI container handles object lifetimes.

3. Types of Dependency Injection

1. Constructor Injection (Most common)
2. Method Injection
3. Property Injection

Example: Constructor Injection

```
public class ProductService
{
    private readonly ILogger<ProductService> _logger;

    public ProductService(ILogger<ProductService> logger)
    {
        _logger = logger;
    }

    public void GetProducts()
    {
        _logger.LogInformation("Fetching products...");
    }
}
```

4. Service Lifetimes in Dependency Injection

.NET Core provides three service lifetimes:

Lifetime	Registration Method	Behavior
Transient	AddTransient<T>()	Creates a new instance every time it's requested.
Scoped	AddScoped<T>()	Creates one instance per request.
Singleton	AddSingleton<T>()	Creates one instance for the application lifetime.

Use Cases

Scenario	Transient	Scoped	Singleton
Stateless Services	✅	❌	❌
Database Context (EF Core)	❌	✅	❌
Global Configuration	❌	❌	✅
Logging, Utilities	✅	❌	❌
Caching Service	❌	❌	✅

5. Registering Services in .NET Core

Services are registered in the Program.cs file using the IServiceCollection.

```
var builder = WebApplication.CreateBuilder(args);

// Register services with different lifetimes
builder.Services.AddTransient<ITransientService, TransientService>();
builder.Services.AddScoped<IScopedService, ScopedService>();
builder.Services.AddSingleton<ISingletonService, SingletonService>();

var app = builder.Build();
app.Run();
```

6. Implementing Dependency Injection in Controllers

DI is automatically applied when injecting dependencies into controllers, middleware, or other services.

```
public class ProductController : Controller
{
    private readonly IProductService _productService;

    public ProductController(IProductService productService)
    {
        _productService = productService;
    }
```

```
    public IActionResult Index()
    {
        _productService.ShowProducts();
        return View();
    }
}
```

7. Dependency Injection Execution Flow

1. App Starts – Services are registered in Program.cs.
2. Request Comes In – Middleware pipeline executes.
3. Controller Instantiated – DI injects registered dependencies.
4. Service Lifetime Applies:
   - Transient → New instance created.
   - Scoped → One instance per request.
   - Singleton → Same instance for all requests.
5. Response Sent – Scoped services are disposed of after the request.

8. Example: Verifying DI Lifetimes

```
public interface IMyService { string GetGuid(); }

public class TransientService : IMyService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => _guid;
}

public class ScopedService : IMyService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => _guid;
}

public class SingletonService : IMyService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => _guid;
}
```

Output Example:

Transient: f4d2a… - Scoped: a1b34… - Singleton: 79ab8…

- Transient changes on every request.
- Scoped stays the same within a request but changes on new requests.
- Singleton remains the same for all requests.

9. Summary

Use Case	Best DI Lifetime
Short-lived services (Logging, Utilities)	`Transient`
Database operations (EF Core, Unit of Work)	`Scoped`
Global services (Caching, Configurations)	`Singleton`

Dependency Injection is a core feature in ASP.NET Core that helps in managing service lifetimes, ensuring maintainability, and improving Testability of applications.

What is CORS in .NET Core?

CORS (Cross-Origin Resource Sharing) in **.NET Core** is a security feature that allows or restricts web applications running at one domain to access resources from another domain. By default, web browsers enforce the **same-origin policy**, which prevents cross-origin requests. CORS is a controlled relaxation of this policy.

builder.Services.AddCors(options => { options.AddPolicy("AllowSpecificOrigins", policy => { policy.WithOrigins("https://example.com") // Allow specific origin .AllowAnyMethod() // Allow any HTTP method (GET, POST, etc.) .AllowAnyHeader(); // Allow any headers }); });

Release vs Debug Mode in .NET Core

In **.NET Core**, there are two main build configurations: **Debug** and **Release**. These configurations determine how the application is compiled and optimized.

1. Debug Mode

Used during development for debugging and testing.

✅ **Characteristics:**

- Includes **debugging symbols** (`.pdb` files) to assist with debugging.
- No optimizations, making it easier to step through code in a debugger.
- More logging and diagnostic information.
- Larger file size and slower execution.

✅ **Use Case:**

- During development and testing.
- Debugging applications in **Visual Studio, VS Code, or Rider**.

✅ **Example: Building in Debug Mode**

```
dotnet build -c Debug
```

2. Release Mode

Used for production deployment with optimizations.

✅ **Characteristics:**

- **Optimized code**: Compiler removes unnecessary metadata and applies performance optimizations.
- **No debugging symbols** by default (though they can be generated separately).
- Smaller executable size and faster execution.
- Disables assertions (`Debug.Assert()`).

✅ **Use Case:**

- When deploying applications to **production**.
- Improving performance and reducing memory usage.

✅ **Example: Building in Release Mode**

```
dotnet build -c Release
```

What are Filters in ASP.NET Core

Filters are used to add cross-cutting concerns, such as Logging, Authentication, Authorization, Exception Handling, Caching, etc., to our application. In ASP.NET Core Applications, Filters allow us to execute cross-cutting logic in the following ways:

1. Before an HTTP request is handled by a controller action method.
2. After an HTTP request is handled by a controller action method.
3. After the response is generated but before it is sent to the client.

In ASP.NET Core MVC, Filters are used to Perform cross-cutting concerns. They are as follows:

- Caching
- Logging
- Error Handling
- Modifying the Result
- Authentication and Authorization, etc

1. What is a JWT Token?

JWT (JSON Web Token) is a compact, self-contained, and secure way to transmit information between parties as a JSON object. It is commonly used for authentication and authorization in web applications and APIs.

✅ Why use JWT?

- Stateless authentication (no need to store sessions).
- Secure (signed using HMAC or RSA).
- Can store user claims (e.g., role, permissions).

A JWT consists of **three parts**, separated by dots

Header.Payload.Signature

Header :-Metadata (Algorithm & Token Type).

Payload:-Contains user claims (e.g., `userId`, `role`).

Signature:-Verifies the token's authenticity.

7. JWT Best Practices

✔ **Use HTTPS** to prevent MITM attacks.
✔ **Keep secret keys secure** (store in environment variables).
✔ **Use short-lived tokens** and refresh tokens for re-authentication.
✔ **Validate JWT tokens properly** (issuer, audience, expiry).

Q: What is JWT and why is it used?
A: "JWT is a secure, compact token used for **authentication** and **authorization** in APIs. It enables stateless authentication, making it scalable and efficient."

Q: How does JWT prevent tampering?
A: "JWT uses a **signature** (HMAC or RSA) to verify that the token has not been modified. If the signature doesn't match, the token is rejected."

Q: What are the common signing algorithms for JWT?
A: "Common algorithms include **HS256** (shared secret, faster) and **RS256** (public-private key, more secure)."

Is a JWT Token Encrypted or Encoded?

A JWT (JSON Web Token) is encoded, not encrypted by default. However, it can be encrypted if needed.

Q: "Is a JWT token encrypted or encoded?"
A: "By default, a JWT is **Base64Url-encoded** and **signed**, but not encrypted. The signature ensures **integrity** but does not hide the data. To make a JWT confidential, we use **JWE (JSON Web Encryption)** to encrypt.

Q: "What is the difference between encoding and encryption?"
A: "Encoding is for **data transformation** and is easily reversible (like Base64). Encryption is for **data security**, making it unreadable without a secret key (like AES or RSA)."

1. What is OAuth?

OAuth (**Open Authorization**) is an **authorization framework** that allows third-party applications to **access resources on behalf of a user** without exposing their credentials. It is commonly used for **single sign-on (SSO)** and **API authorization**.

✅ **Why use OAuth?**

- Secure access to resources **without sharing passwords**.
- Supports **third-party authentication** (Google, Facebook, GitHub, etc.).
- Enables **token-based authentication** for APIs.
- **User control** over granted permissions.

6. Advantages of OAuth

✔ No password sharing – Users don't provide passwords to third-party apps.
✔ Single Sign-On (SSO) – One login for multiple services.
✔ Fine-grained access control – Users can grant/deny permissions.
✔ Stateless authentication – Uses access tokens instead of sessions.

Q: What is OAuth and how does it work?
A: "OAuth is an **authorization framework** that allows users to grant third-party applications **limited access to their resources** without sharing credentials. It works by issuing **access tokens** after user consent."

Q: What is the difference between OAuth and JWT?
A: "OAuth is a **protocol** for authentication & authorization, while JWT is a **token format** used within authentication systems, including OAuth."

Q: What is the most secure OAuth flow?
A: "The **Authorization Code Grant** is the most secure, as it does not expose tokens in the browser."

Q: What is the role of `Startup.cs` in .NET Core?
A: "`Startup.cs` is used in .NET Core 5 and earlier to configure services (`ConfigureServices`) and middleware (`Configure`). In .NET 6+, it is merged into `Program.cs` for a simplified hosting model."

Q: What is the difference between `ConfigureServices` and `Configure`?
A: "`ConfigureServices` is for registering dependencies (DI), while `Configure` sets up middleware for request processing."

✅ `Startup.cs` was used for **service registration & middleware setup** in .NET Core 5 and earlier.

✅ In **.NET 6+, `Startup.cs` is removed**, and all configurations move to `Program.cs`.

✅ Understanding `Startup.cs` helps in maintaining **legacy .NET Core projects**.

Conventional Routing vs Attribute Routing in .NET Core

Routing in ASP.NET Core determines how incoming HTTP requests are mapped to controllers and actions. There are two types of routing:

1. **Conventional Routing** (centralized, pattern-based)
2. **Attribute Routing** (decentralized, applied at the controller/action level)

Typically used in MVC-style applications with controllers.

✅ **Pros**
✔ Easy to manage centrally.
✔ Good for MVC applications.

❌ **Cons**
✖ Less flexible for complex APIs.
✖ Becomes hard to manage with many routes.

2. Attribute Routing

📌 **Definition**:

- Uses **attributes** directly on controller actions.
- More **flexible and readable** than conventional routing.
- Mostly used in **RESTful APIs**.

[HttpGet] // Matches: GET /api/products public IActionResult GetAll() => Content("All Products");

Q: What is the difference between Conventional Routing and Attribute Routing?

A: "Conventional Routing is **defined centrally** in `Program.cs` and follows a **pattern-based approach**, mainly used in MVC apps. Attribute Routing is **defined using attributes** on controllers and actions, making it **more flexible** and ideal for RESTful APIs."

🔹 Final Thoughts

✅ **Use Conventional Routing** for MVC-based applications.
✅ **Use Attribute Routing** for RESTful APIs where flexibility is needed.
✅ **You can mix both** in a project for different needs.

Code-First vs Database-First Approach in Entity Framework Core

Entity Framework Core (EF Core) provides two primary ways to interact with a database:

1️⃣ Code-First Approach → Define the model (C# classes), and EF Core creates the database.

2️⃣ Database-First Approach → Use an existing database, and EF Core generates model classes.

📌 **Definition**:

- In the **Code-First approach**, you create **C# entity classes** first, and EF Core generates the **database schema**.
- Migrations are used to keep the database updated.

📌 **When to Use?**
✅ When starting a new project without an existing database.
✅ When you want full control over database design using C# classes.

✅ **Advantages of Code-First**
✔ Full control over entity design.
✔ Easy to evolve the database using **migrations**.
✔ Good for Agile development.

❌ **Disadvantages**
✖ Not ideal for large, complex databases.
✖ Risk of accidental schema changes.

2. Database-First Approach

📌 **Definition**:

- In the **Database-First approach**, you start with an **existing database**, and EF Core **generates the entity classes**.
- Reverse engineering is used to scaffold models from the database.

📌 **When to Use?**
✅ When working with an **existing** database.
✅ When the database is **managed separately** by DBAs.

✅ **Advantages of Database-First**
✔ Ideal for working with **existing** databases.
✔ Avoids manual creation of entity classes.
✔ Ensures consistency with the DB schema.

❌ **Disadvantages**
✖ Harder to modify the database structure from the code.
✖ Requires **manual re-scaffolding** when the database changes.

SOAP vs REST APIs with HTTP Verbs & Status Codes

APIs (Application Programming Interfaces) allow communication between different systems. The two most common types of APIs are **SOAP (Simple Object Access Protocol)** and **REST (Representational State Transfer)**.

🔹 **REST is more commonly used today** due to its simplicity and efficiency

HTTP Verbs:-GET, POST, PUT, DELETE, PATCH

Communicate through json format hence its faster

5️⃣ When to Use SOAP vs REST?

✅ **Use SOAP when:**

- High **security** is needed (**banking, payment systems**).
- Transactions require **ACID compliance** (e.g., financial operations).
- **Enterprise-level applications** with strict protocols.

✅ **Use REST when:**

- Building **web services, mobile apps, and microservices**.
- Performance and **scalability** are important.
- **JSON-based lightweight communication** is preferred

Q: What is the difference between SOAP and REST APIs?
A: "SOAP is a protocol that uses **XML** and supports **multiple transport methods** like HTTP and SMTP, making it **secure but slow**. REST is an **architectural style** using **JSON/XML over HTTP**, making it **lightweight and scalable**."

Q: Which HTTP verbs are used in REST APIs?
A: "**GET, POST, PUT, PATCH, DELETE** – GET fetches data, POST creates a resource, PUT updates completely, PATCH updates partially, and DELETE removes a resource."

Q: What HTTP status codes are commonly used in REST APIs?
A: "200 for success, 201 for resource creation, 400 for bad requests, 401 for unauthorized access, 404 for not found, and 500 for internal server errors."

Q: When would you use SOAP instead of REST?
A: "When **high security, reliability, and transactional consistency** are required, such as in **financial services, payment systems, and telecom billing**."

5️⃣ Interview Tips 💡

Q: Can we create a SOAP API in .NET Core?
A: "Yes, but .NET Core does not support WCF. Instead, we can use the `SoapCore` NuGet package to create SOAP APIs in .NET Core."

Q: How do you expose a SOAP endpoint in .NET Core?
A: "Use `SoapCore` to register the service and expose a `.svc` endpoint in `Program.cs`."

Q: What is `basicHttpBinding` in WCF?
A: "`basicHttpBinding` is the most common SOAP binding in WCF that works over HTTP and is compatible with non-.NET clients."

Q: Which is better: REST or SOAP?
A: "REST is preferred for web and mobile apps due to its lightweight nature. SOAP is better for high-security enterprise applications like banking."

1️⃣ What is Reflection?

Reflection in .NET Core is a **runtime feature** that allows **introspection** and **modification** of types, methods, fields, properties, and assemblies. It enables **dynamically accessing metadata** and invoking members of a class at runtime.

- **Namespace:** `System.Reflection`
- **Use Cases:**
  - Inspecting types, properties, and methods at runtime
  - Dynamically invoking methods (e.g., for plugins, dependency injection)
  - Creating objects dynamically without knowing their type at compile-time
  - Analyzing assemblies, classes, and members

6️⃣ Summary

✅ **Reflection allows runtime type inspection and method invocation**.
✅ **Useful for dependency injection, ORM, serialization, and plugin systems**.
✅ **Avoid Reflection in performance-critical scenarios**; use compiled expressions instead.
✅ **Common in ASP.NET Core for dynamic object creation and service loading**.

```
using System;
using System.Reflection;

class Program
{
    static void Main()
    {
        Type type = typeof(Person);
        Console.WriteLine("Class Name: " + type.Name);
    }
}

class Person
{
    public string Name { get; set; }
    public void SayHello() => Console.WriteLine("Hello!");
}
```