🔷 SECTION 2: DEPENDENCY INJECTION (THEORY – IN DEPTH) 🔷

1️⃣ What is Dependency Injection?

🧠 Imagine a TV 📺 .

> You don't build speakers inside the TV. You plug speakers from outside.

Similarly:

- A class should not **create its own dependencies**
- Dependencies should be **provided from outside**

That's Dependency Injection.

🎯 Dependency Injection is a **design pattern** where:

- Objects do not create their dependent objects
- Dependencies are provided by an **IoC container**
- Promotes loose coupling, testability, and maintainability

📌 ASP.NET Core has a built-in DI container.

🔹 **"Dependency Injection is a technique where an object receives its dependencies from an external source rather than creating them internally."**

2️⃣ Why Dependency Injection is Needed?

🧠 If you hard-code dependencies:

- Code becomes rigid
- Testing becomes difficult
- Changes break many places

🎯 DI helps to:

- Replace implementations easily
- Mock dependencies in unit testing
- Follow SOLID principles (especially D)

🔁 Q: Which SOLID principle does DI support?

✓ Dependency Inversion Principle

3️⃣ Dependency Injection in ASP.NET Core

🧠 ASP.NET Core automatically:

- Creates objects
- Injects dependencies
- Manages object lifetime

🎯 ASP.NET Core uses a built-in IoC container that:

- Registers services at startup
- Resolves dependencies at runtime
- Supports constructor injection by default

📌 Constructor injection is recommended & default.

4️⃣ Types of Dependency Injection

1️⃣ **Constructor Injection** (MOST IMPORTANT)

🧠 Give everything a class needs at the time of creation.

🎯 Interview Perspective

- Preferred approach
- Makes dependencies explicit
- Prevents partially initialized objects

📌 ASP.NET Core natively supports only constructor injection.

2️⃣ **Property Injection**

- Not supported by default
- Can cause null dependencies
- Not recommended

3️⃣ **Method Injection**

- Used rarely
- Explicit passing of dependencies per method

🔁 Q: Does ASP.NET Core support property injection?

❌ No (out of the box)

5️⃣ **Service Lifetimes (VERY IMPORTANT)**

This is where most candidates fail ❌

🔵 **Transient** 🔵	🟢 **Scoped (MOST USED)** 🟢	🔴 **Singleton** 🔴
🧠 New object **every time**. 🎯 Interview Perspective • Created each time requested • Lightweight, stateless services 📌 Example: • Logging • Validation	🧠 One object **per request**. 🎯 Interview Perspective • Same instance within HTTP request • New instance for each request 📌 Example: • DbContext	🧠 One object for **entire application lifetime**. 🎯 Interview Perspective • Created once • Shared across all requests • Must be thread-safe 📌 Example: • Configuration • Caching services

🚨 Q: Can Singleton depend on Scoped service? ❌ NO (Captive dependency)

6️⃣ Service Registration Methods

```
services.AddTransient<IEmailService, EmailService>();
services.AddScoped<IOrderService, OrderService>();
services.AddSingleton<ICacheService, CacheService>();
```

📌 **Interface-based registration is best practice.**

7️⃣ Dependency Resolution Flow (INTERVIEW GOLD)

ASP.NET Core:

- Sees controller
- Looks at constructor
- Resolves dependencies from container
- Injects them

🎯 Interview Perspective

- Resolution happens **at runtime**
- Throws exception if dependency not registered
- Uses constructor with **maximum resolvable parameters**

🔁 Q: What if multiple constructors exist?

✓ Uses the one with most resolvable parameters

8️⃣ DI + Middleware vs Controllers

- Controllers → constructor injection
- Middleware → constructor + `Invoke` method injection

📌 Middleware supports **method injection**, controllers don't.

9️⃣ Common DI Interview Mistakes 🚨

- ❌ Injecting DbContext as Singleton
- ❌ Using Service Locator pattern
- ❌ Too many constructor dependencies
- ❌ Injecting concrete classes instead of interfaces

🔟 DI & Testing (IMPORTANT)

🎯 DI allows:

- Mocking dependencies
- Isolated unit tests
- Cleaner test setup

📌 Without DI → testing is painful.

🧠 FINAL INTERVIEW SUMMARY (MEMORIZE)

"ASP.NET Core has a built-in dependency injection container that supports constructor injection and manages object lifetimes using transient, scoped, and singleton services to promote loose coupling, testability, and maintainability."

🔷 HANDS-ON SECTION 3: DEPENDENCY INJECTION (PRACTICAL) 🔷

🧱 STEP 1: Create a Demo Project

🧱 STEP 2: Create Interfaces & Implementations

- 🔹 Interface

```
public interface IMessageService
{
    string GetMessage();
}
```

- 🔹 Implementation

```
public class EmailMessageService : IMessageService
{
    public string GetMessage()
    {
        return "Message sent via Email Service";
    }
}
```

🧱 STEP 3: Register the Service in `Program.cs`

This is where we tell the IoC container how to resolve our dependency.

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

builder.Services.AddControllers();

// Register dependency
builder.Services.AddTransient<IMessageService, EmailMessageService>();

var app = builder.Build();

app.MapControllers();

app.Run();
```

📌 Interview Explanation:

We register our service with a lifetime (Transient, Scoped, Singleton). The IoC container now knows that whenever a controller or class needs an `IMessageService`, it should create an instance of `EmailMessageService`.

🧱 STEP 4: Inject Service into Controller

Example Controller

```
[ApiController]
[Route("api/[controller]")]
public class NotificationController : ControllerBase
{
    private readonly IMessageService _messageService;

    // Constructor Injection
    public NotificationController(IMessageService messageService)
    {
        _messageService = messageService;
    }

    [HttpGet]
    public IActionResult GetMessage()
    {
        return Ok(_messageService.GetMessage());
    }
}
```

Output

```
GET /api/notification
→ "Message sent via Email Service"
```

📌 Interview Talk Track

Here, the framework automatically injects an instance of `EmailMessageService` into the controller constructor. This shows constructor injection — the default and most recommended injection pattern in ASP.NET Core.

🧱 STEP 5: Demonstrate Lifetimes

Add 3 services:

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

Implementations

```
public class TransientGuidService : IGuidService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => $"Transient: {_guid}";
}

public class ScopedGuidService : IGuidService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => $"Scoped: {_guid}";
}

public class SingletonGuidService : IGuidService
{
    private readonly string _guid = Guid.NewGuid().ToString();
    public string GetGuid() => $"Singleton: {_guid}";
}
```

Register Them

```
builder.Services.AddTransient<TransientGuidService>();
builder.Services.AddScoped<ScopedGuidService>();
builder.Services.AddSingleton<SingletonGuidService>();
```

Controller to Test Lifetime

```
[ApiController]
[Route("api/[controller]")]
public class LifetimeController : ControllerBase
{
    private readonly TransientGuidService _transient;
    private readonly ScopedGuidService _scoped;
    private readonly SingletonGuidService _singleton;

    public LifetimeController(
        TransientGuidService transient,
        ScopedGuidService scoped,
        SingletonGuidService singleton)
    {
        _transient = transient;
        _scoped = scoped;
```

```
        _singleton = singleton;
    }


    [HttpGet]
    public string Get()
    {
        return $"{_transient.GetGuid()} \n{_scoped.GetGuid()}
\n{_singleton.GetGuid()}";
    }
}
```

- 🔹 Test Output (Refresh twice in browser)

First request

```
Transient: a1b2...
Scoped: b3c4...
Singleton: c4d5...
```

Second request

```
Transient: NEW GUID
Scoped: NEW GUID
Singleton: SAME GUID
```

- 🔹 Interview Explanation
  - Transient → new instance every time
  - Scoped → one instance per HTTP request
  - Singleton → same instance across all requests

You can explain this demo in 20 seconds and it's one of the best ways to show practical DI knowledge in interviews.

🧱 STEP 6: Interface Injection for Testing (Mock Example)

Example:

```
public class SmsMessageService : IMessageService
{
    public string GetMessage() => "Message sent via SMS Service";
}
```

You can easily swap the implementation in `Program.cs`:

```
builder.Services.AddTransient<IMessageService, SmsMessageService>();
```

No code changes in controller!

📌 This demonstrates loose coupling and testability.

📌 Interview Summary Line:

“Using DI, I can switch service implementations (e.g., Email vs SMS) by changing a single line in Startup without modifying controller logic, which supports open/closed principle.”

🧱 STEP 7: Middleware DI Example (Bonus)

```
public class LogMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IMessageService _messageService;

    public LogMiddleware(RequestDelegate next, IMessageService messageService)
    {
        _next = next;
        _messageService = messageService;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"DI Middleware says:
{_messageService.GetMessage()}");
        await _next(context);
    }
}
```

Register in `Program.cs`:

```
app.UseMiddleware<LogMiddleware>();
```

📌 Middleware supports both constructor and method injection.

🧠 FINAL INTERVIEW ANSWER (MEMORIZE)

“ASP.NET Core’s built-in DI container lets us register dependencies with lifetimes — transient, scoped, and singleton. Dependencies are automatically injected via constructor, which promotes testability, loose coupling, and adherence to SOLID principles.”

✅ YOU NOW MASTER

✓ Constructor injection demo ✓ Lifetime behaviour demo ✓ Interface swapping demo ✓ Middleware injection ✓ IoC container concept