
✅ Dependency Injection (DI) in ASP.NET Core

Dependency Injection is a design pattern where **an object (service)** gets the **dependencies it needs** from an external system (DI Container) **instead of creating them itself**.

Why DI?

- Reduces tight coupling
- Improves unit testing
- Promotes clean architecture
- Allows plugging different implementations easily

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

✅ 1. Service Lifetime

ASP.NET Core provides three main lifetimes for service registration:

A. AddTransient

- **New object every time** it is requested.
- Lightweight, stateless services.

Example:

services.AddTransient<IGreetingService, GreetingService>();

If controller calls it three times → **3 different objects!**

When to use?

- Logging adapters
- Helper classes
- Calculation/formatting classes

B. AddScoped

- **One object per HTTP Request**.

- Same instance is shared inside **request pipeline**.

Example:

services.AddScoped<IOrderService, OrderService>();

When to use?

- DB-related services
- Business logic services
- Unit of Work patterns

C. AddSingleton

- **Created once only** → at application start.
- Used forever in the app.

Example:

services.AddSingleton<IAppCache, AppCache>();

When to use?

- App configuration data
- In-memory caching
- Services that must maintain global state

⚠️ **DO NOT use with DbContext**
It will cause concurrency issues because DbContext is not thread-safe.

⚡ **Practical Example: Observing Lifetimes**

Step 1: Create Interfaces & Classes

IGuidService.cs

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

```
}
```

Implementations

Transient

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

Scoped

```
public class ScopedGuidService : IGuidService
{
    private readonly Guid _id = Guid.NewGuid();
    public string GetGuid() => _id.ToString();
}
```

Singleton

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

Step 2: Register in Program.cs

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

Step 3: Inject into a Controller

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

    public GuidDemoController(
        TransientGuidService transient,
        ScopedGuidService scoped,
        SingletonGuidService singleton)
    {
        _transient = transient;
        _scoped = scoped;
        _singleton = singleton;
    }

    [HttpGet]
    public IActionResult Get()
    {
        return Ok(new {
            Transient = _transient.GetGuid(),
            Scoped = _scoped.GetGuid(),
            Singleton = _singleton.GetGuid()
```

```
    });
  }
}
```

If you refresh the browser:

Lifetime	Same Request	New Request
Transient	new	new
Scoped	same	new
Singleton	same	same

✅ 2. Constructor Injection

Constructor Injection is the **most common DI technique**.

Why?

- Clear dependency declaration
- Avoids “hidden” dependencies
- Safe and testable

Example:

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

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

```
    [HttpGet]
    public IActionResult GetAll()
    {
        return Ok(_productService.GetProducts());
    }
}
```

ASP.NET Core automatically creates the object and injects the dependency.

✅ 3. Interface-Driven Development

Interface-driven development means:

- **Code depends on abstractions, not concrete implementations.**
- Easy to replace the implementation (e.g., for unit testing).
- Supports loose coupling and clean architecture.

Example

Step 1: Create Interface

```
public interface IEmailSender
{
    void SendEmail(string to, string subject, string body);
}
```

Step 2: Create Implementation

```
public class SmtpEmailSender : IEmailSender
{
    public void SendEmail(string to, string subject, string body)
    {
        // SMTP code here
    }
```

```
}
```

Step 3: Register in DI

```
services.AddScoped<IEmailSender, SmtpEmailSender>();
```

Step 4: Use in Controller

```
public class AccountController : ControllerBase
{
    private readonly IEmailSender _email;

    public AccountController(IEmailSender email)
    {
        _email = email;
    }

    [HttpPost("register")]
    public IActionResult Register(string userEmail)
    {
        _email.SendEmail(userEmail, "Welcome", "Thank you for registering!");
        return Ok("Email Sent");
    }
}
```

If tomorrow you want SendGrid or Twilio email → change only DI registration.

🎯 **Summary Table**

Lifetime	Instance Created	Best Used For
Transient	Every request	Lightweight, stateless

Lifetime	Instance Created	Best Used For
Scoped	Per HTTP Request	Business logic, DB operations
Singleton	Once per app	Cache, configuration

🎓 Final Note

DI + Interfaces =

- ✔ Cleaner architecture
- ✔ Easy maintenance
- ✔ Testable code
- ✔ Professional enterprise coding
