Dependency Injection

5 Mistakes You Should Never Make

by Sanuja Tharinda

by Sanuja Tharinda

Quick Recap

- **Dependency Injection** (DI) is the process of passing objects or services (dependencies) into a class, instead of letting the class create them internally.

- **Key Benefits** of DI are**,**
    - Loosely Coupled Code
    - Easier Unit Testing
    - Better Maintainability

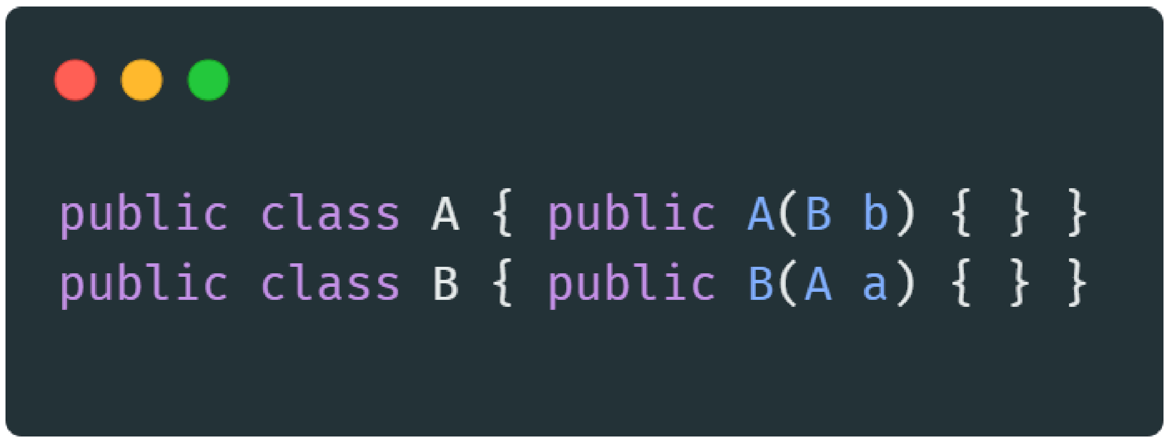
- But many developers run **into issues** when using it. Let's explore some common problems and how to fix them!

Swipe to next slide

by Sanuja Tharinda

Circular Dependancies

- A **circular dependency** happens when two or more services depend on each other, creating an infinite loop.



```
public class A { public A(B b) { } }
public class B { public B(A a) { } }
```

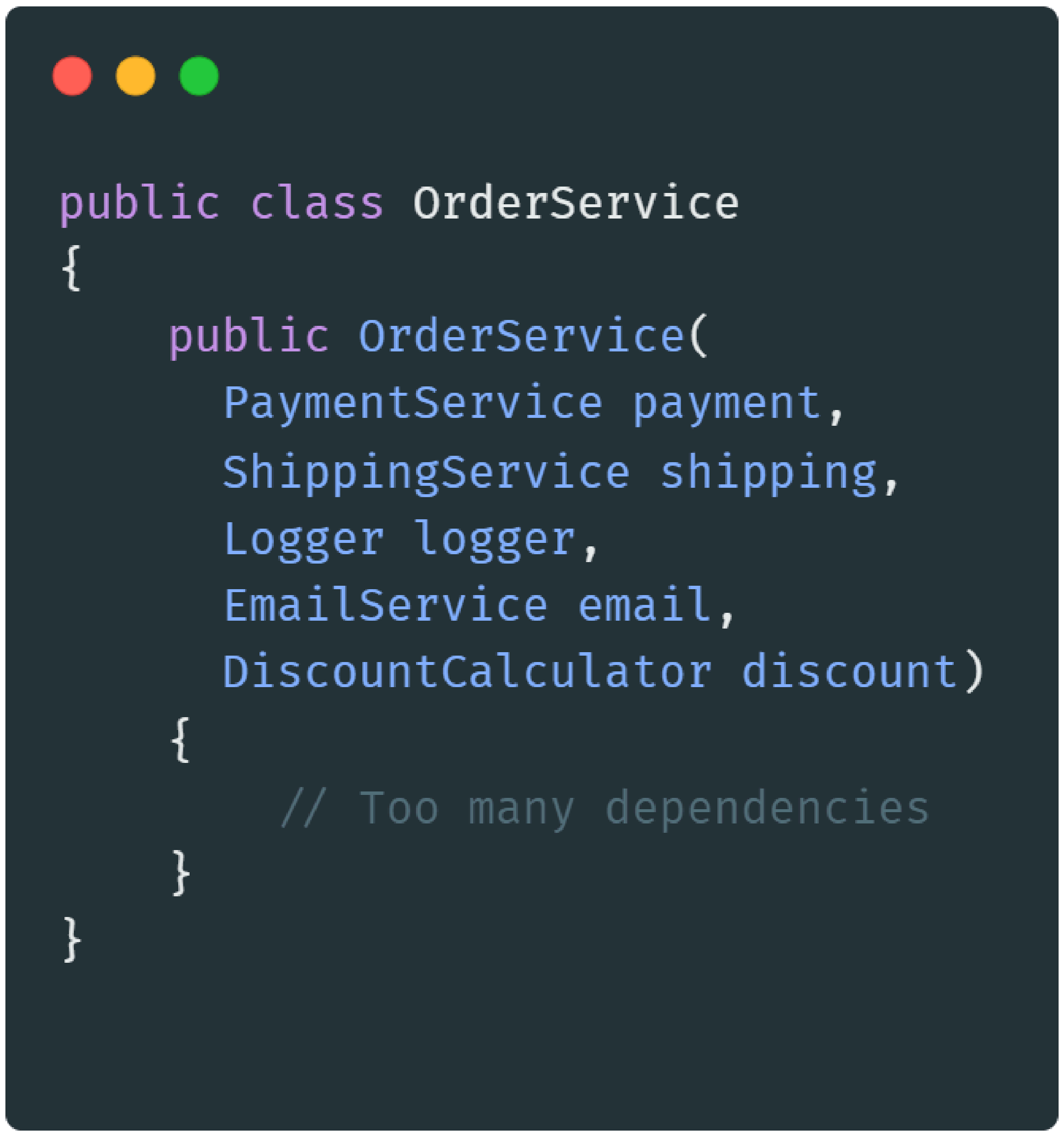
- ✓ **Solution**
  - **Refactor the design** to avoid circular dependencies.
  - Use **Lazy<T>** to delay object creation
  - Use a **factory pattern** to create objects when needed.

Swipe to next slide

by Sanuja Tharinda

Too Many Dependencies in a Single Class

- When a class has **too many injected dependencies**, it usually indicates that the class is doing too much (violating Single Responsibility Principle).



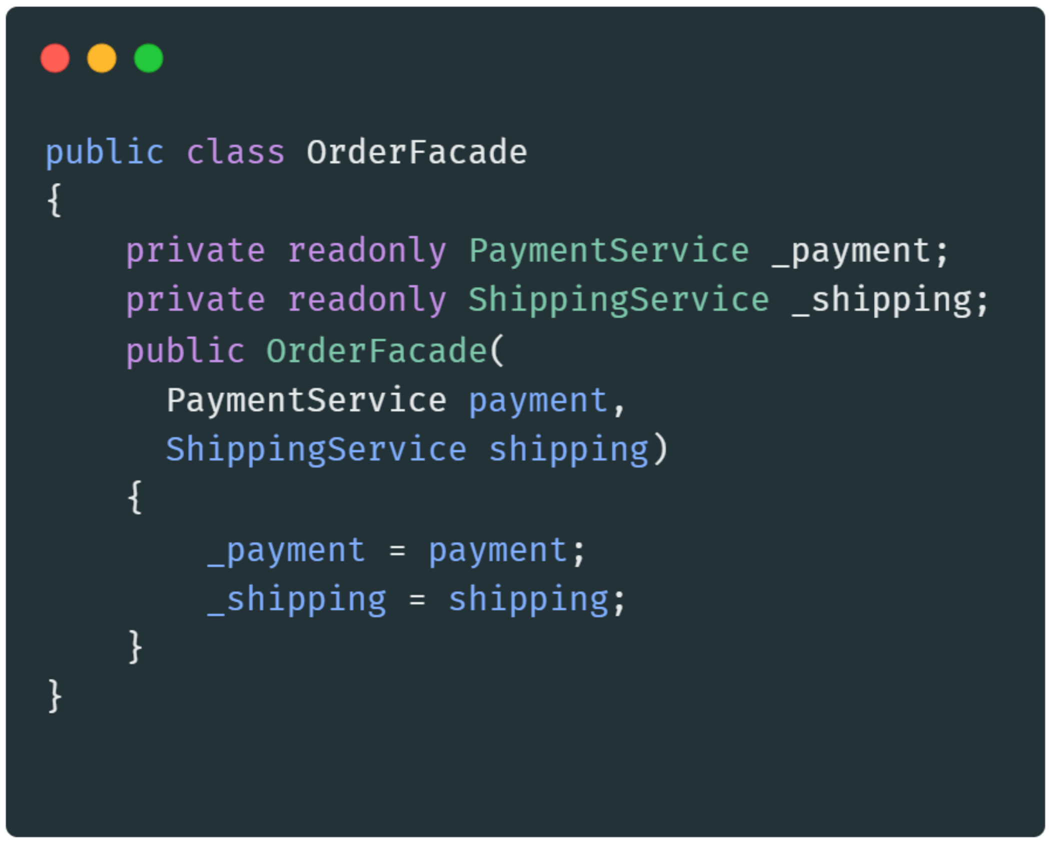
```
public class OrderService
{
    public OrderService(
      PaymentService payment,
      ShippingService shipping,
      Logger logger,
      EmailService email,
      DiscountCalculator discount)
    {
        // Too many dependencies
    }
}
```

Swipe to next slide

by Sanuja Tharinda

Too Many Dependencies in a Single Class

- This makes **unit testing** hard and increases coupling between services.
- ✓ **Solution**
  - Refactor the class by breaking it into **smaller services**.
  - Use **Facade** or **Mediator** patterns to group related dependencies.



```
public class OrderFacade
{
    private readonly PaymentService _payment;
    private readonly ShippingService _shipping;
    public OrderFacade(
      PaymentService payment,
      ShippingService shipping)
    {
        _payment = payment;
        _shipping = shipping;
    }
}
```

Swipe to next slide →

by Sanuja Tharinda

Lifetime Mismatches in DI

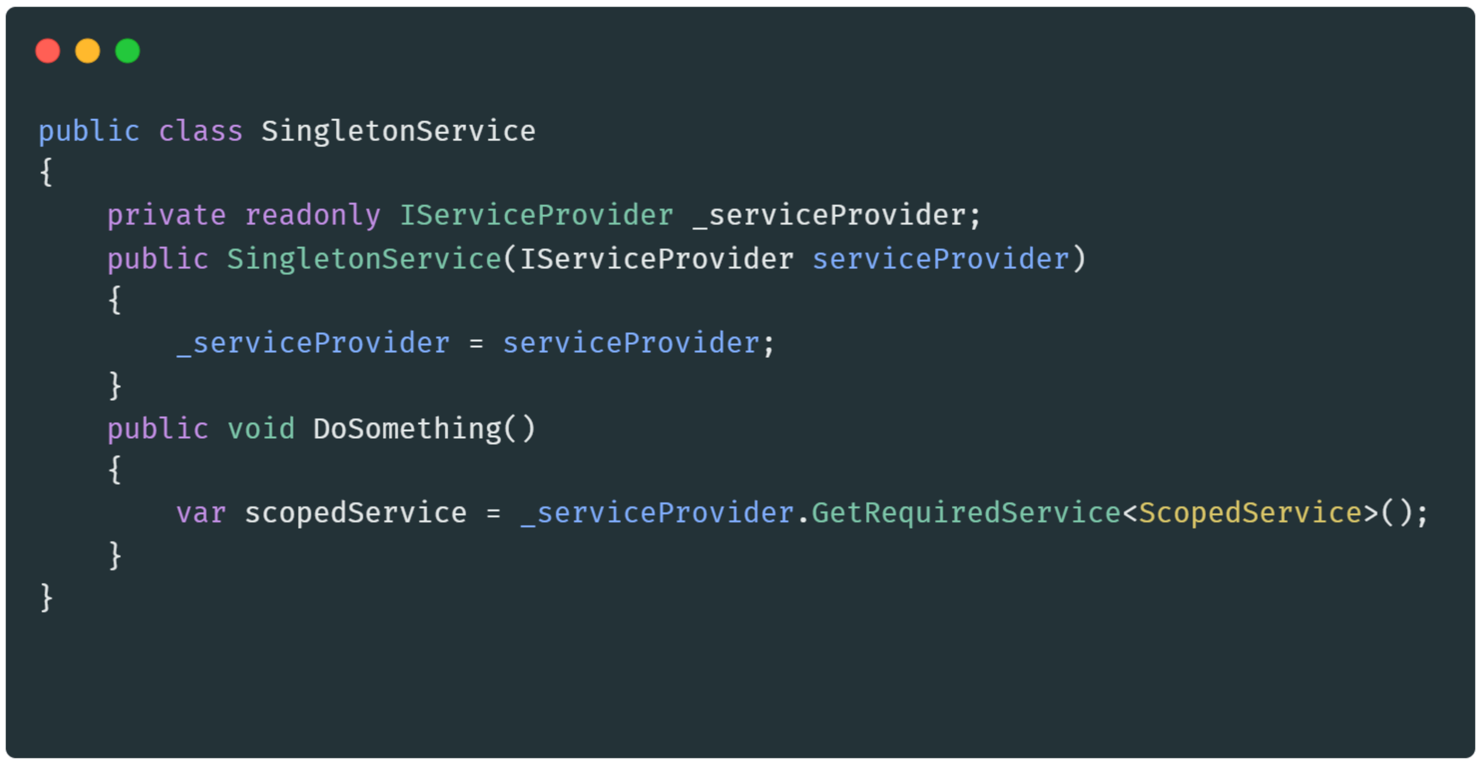
- In ASP.NET Core, dependencies have different lifetimes:
    - **Transient** – Created every time it’s requested.
    - **Scoped** – Created once per request.
    - **Singleton** – Created once for the entire application.
- Incorrect mismatches in DI can lead to **issues** or **errors**.
- Example:
    - Injecting a scoped service into a singleton can cause unexpected behavior.
    - Scoped Service cannot be resolved inside a Singleton.

Swipe to next slide

by Sanuja Tharinda

Lifetime Mismatches in DI

- ✓ **Solution**
  - Instead of injecting ScopedService directly, inject IServiceProvider and resolve it manually



```
public class SingletonService
{
    private readonly IServiceProvider _serviceProvider;
    public SingletonService(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }
    public void DoSomething()
    {
        var scopedService = _serviceProvider.GetRequiredService<ScopedService>();
    }
}
```

  - Alternatively, refactor your dependencies to avoid lifetime mismatches.

Swipe to next slide

by Sanuja Tharinda

Overusing Singleton Services

- While **Singletons** are useful, **overusing them can cause performance issues** and **state-related bugs**.

- If a singleton service **holds state**, it can cause **unexpected behavior** across multiple requests.

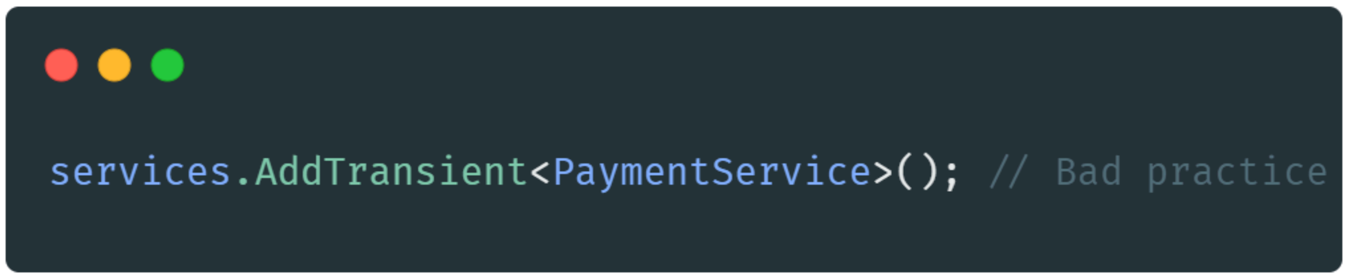
- ✓ **Solution**
  - If you **must** use a singleton, **make it stateless** or use a **ConcurrentDictionary** to manage state safely.

Swipe to next slide →

by Sanuja Tharinda

Injecting Concrete Classes Instead of Interfaces

- DI is **meant to promote loose coupling**, but sometimes developers inject **concrete classes** instead of interfaces.



```
services.AddTransient<PaymentService>(); // Bad practice
```

- ✓ **Solution**
    - Always inject interfaces instead of concrete classes.
    - This makes it **easier to mock dependencies** for unit testing and follow **Dependency Inversion Principle**.

Swipe to next slide

What’s your biggest challenge with DI?

by Sanuja Tharinda
Software Engineer

Let’s Connected