**Dependency Injection**

**Vorteile von Dependency Injection**

**1. Lose Kopplung**

Statt dass eine Klasse selbst ihre Abhängigkeiten erstellt (new), bekommt sie sie von außen „eingespritzt“. Dadurch hängt sie **nicht mehr direkt** von konkreten Implementierungen ab, sondern nur von Interfaces oder Basisklassen.

public class BestellService

{

private readonly IEmailService \_emailService;

public BestellService(IEmailService emailService)

{

\_emailService = emailService;

}

}

**2. Einfaches Testen**

Durch DI kann man im Testkontext problemlos **Mocks oder Stubs** einspeisen. Das macht Unit-Tests unabhängig von Datenbanken oder realen Services.

var mockEmail = new Mock<IEmailService>();

var service = new BestellService(mockEmail.Object);

**3. Zentrale Konfiguration**

Die Verwaltung aller Abhängigkeiten erfolgt **an einer zentralen Stelle** (z. B. Program.cs oder Startup.cs), was den Überblick und die Kontrolle erleichtert.

**4. Einheitliches Lebensdauermanagement**

Man kann gezielt festlegen, wie lange ein Objekt leben soll:

* **Singleton**: eine Instanz für die gesamte Anwendung
* **Scoped**: eine Instanz pro Anforderung (z. B. HTTP-Request)
* **Transient**: neue Instanz bei jedem Zugriff

**5. Erweiterbarkeit**

Man kann Implementierungen leicht austauschen, z. B. für neue Anforderungen oder andere Datenquellen – **ohne Änderungen an der Kernlogik**.

Beispiel: Austausch einer FileLogger- durch eine DatabaseLogger-Implementierung – durch Änderung an einer einzigen Stelle.

**Voraussetzungen**

NuGet-Pakete:

* + Microsoft.Extensions.Hosting
  + Microsoft.Extensions.DependencyInjection
  + Microsoft.EntityFrameworkCore
  + Microsoft.EntityFrameworkCore.SqlServer