Dependency Injection (DI) und Dependency Injection Container

Agenda

1. Einführung in Dependency Injection
2. Vorteile von Dependency Injection
3. Arten von Dependency Injection
4. Einführung in DI-Container
5. Beispiel: Einfacher DI-Container
6. Fortgeschrittene Konzepte in DI
7. Lebenszyklus-Management von Abhängigkeiten
8. Fazit und Best Practices

Was ist Dependency Injection?

- **Definition**: Dependency Injection (DI) ist ein Entwurfsmuster, bei dem Objekte ihre Abhängigkeiten von außen erhalten (anstatt sie selbst zu erstellen).
- **Hauptziel**: Erhöht die Testbarkeit und Flexibilität des Codes.

Beispiel:

```
public class NotificationManager
{
    private readonly IMessageService _messageService;

    public NotificationManager(IMessageService messageService)
    {
        _messageService = messageService;
    }

    public void SendNotification(string message)
    {
        _messageService.SendMessage(message);
    }
}
```

Vorteile von Dependency Injection

1. **Lose Kopplung**: Objekte sind weniger abhängig von spezifischen Implementierungen.
2. **Testbarkeit**: Ermöglicht einfaches Mocking und Unit-Tests.
3. **Erweiterbarkeit**: Erleichtert den Austausch von Abhängigkeiten ohne Codeänderungen.
4. **Wiederverwendbarkeit**: Reduziert redundanten Code und erhöht die Wiederverwendbarkeit von Komponenten.

Arten von Dependency Injection

Constructor Injection

- Abhängigkeiten werden über den Konstruktor bereitgestellt.
- Am häufigsten verwendet.

```
public class MyClass
{
    private readonly IDependency _dependency;

    public MyClass(IDependency dependency)
    {
        _dependency = dependency;
    }
}
```

Property Injection

- Abhängigkeiten werden über Eigenschaften bereitgestellt.

```
public class MyClass
{
    public IDependency Dependency { get; set; }
}
```

Method Injection

- Abhängigkeiten werden als Parameter einer Methode bereitgestellt.

```
public class MyClass
{
    public void DoWork(IDependency dependency)
    {
        // Nutzung von dependency
    }
}
```

Einführung in DI-Container

- **Definition**: Ein DI-Container verwaltet automatisch die Erstellung und Auflösung von Abhängigkeiten.
- **Funktionsweise**:
    i. Abhängigkeiten werden registriert.
    ii. Der Container löst die Abhängigkeiten automatisch auf, wenn sie benötigt werden.

Vorteile:

- Automatisiert die Verwaltung von Abhängigkeiten.
- Unterstützt komplexe Abhängigkeitsstrukturen.
- Ermöglicht einfaches Lebenszyklus-Management von Objekten.

Beispiel eines einfachen DI-Containers

- **Schritt 1**: Registrierung von Abhängigkeiten
- **Schritt 2**: Auflösung von Abhängigkeiten

Beispiel:

```
public class SimpleContainer
{
    private Dictionary<Type, Type> _typeMappings = new Dictionary<Type, Type>();

    public void Register<TInterface, TImplementation>()
    {
        _typeMappings[typeof(TInterface)] = typeof(TImplementation);
    }

    public TInterface Resolve<TInterface>()
    {
        return (TInterface)Resolve(typeof(TInterface));
    }

    private object Resolve(Type type)
    {
        Type implementationType = _typeMappings[type];
        var constructorInfo = implementationType.GetConstructors()[0];
        var parameters = constructorInfo.GetParameters();

        if (parameters.Length == 0)
        {
            return Activator.CreateInstance(implementationType);
        }
        else
        {
            var parameterImplementations = new List<object>();
            foreach (var parameter in parameters)
            {
                parameterImplementations.Add(Resolve(parameter.ParameterType));
            }

            return constructorInfo.Invoke(parameterImplementations.ToArray());
        }
    }
}
```

Fortgeschrittene Konzepte in Dependency Injection

Lazy Initialization

- Verzögertes Erstellen von Abhängigkeiten bis zum Zeitpunkt der ersten Nutzung.
- Beispiel: `Lazy<T>` in C#.

Scoped und Transient Abhängigkeiten

- **Transient**: Eine neue Instanz wird bei jeder Auflösung erstellt.
- **Singleton**: Es gibt nur eine Instanz, die während der gesamten Laufzeit verwendet wird.
- **Scoped**: Eine Instanz wird pro Anfrage/Scope erstellt.

Lebenszyklus-Management von Abhängigkeiten

- **Transient**:
  - Jedes Mal, wenn eine Abhängigkeit benötigt wird, wird eine neue Instanz erstellt.
- **Singleton**:
  - Der Container erstellt nur eine einzige Instanz, die wiederverwendet wird.
- **Scoped**:
  - Eine Instanz wird innerhalb eines bestimmten Scopes (z. B. einer HTTP-Anfrage) erstellt und verwendet.

Verwendung eines fortgeschrittenen DI-Containers

Hier ist ein Beispiel für die Verwendung des **Microsoft.Extensions.DependencyInjection** Containers:

```
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    static void Main(string[] args)
    {
        var serviceProvider = new ServiceCollection()
            .AddTransient<IMessageService, EmailMessageService>()
            .BuildServiceProvider();

        var notificationManager = serviceProvider.GetService<NotificationManager>();
        notificationManager.SendNotification("Hello, DI with a container!");
    }
}
```

- **DI-Container**: Verwaltet den Lebenszyklus und die Auflösung von Abhängigkeiten.
- **AddTransient**, **AddSingleton**, **AddScoped**: Lebenszyklusoptionen.

Fazit und Best Practices

- **Lose Kopplung**: DI ermöglicht flexible und modularisierte Software-Architekturen.
- **Testbarkeit**: Abhängigkeiten können einfach gemockt werden, was Unit-Tests erleichtert.
- **Best Practices**:
    i. Verwende Constructor Injection als Standard.
    ii. Halte die Abhängigkeitsstruktur einfach.
    iii. Vermeide Service Locator Pattern, wenn möglich.
    iv. Nutze einen DI-Container zur Verwaltung komplexer Abhängigkeitsstrukturen.