Dependency Injection (DI) WPF

Inhalt

1 Einführung

Bei umfangreicheren Anwendungen werden oft einzelne Objekte (wie z.B. die DbContext-Instanz) in mehreren Klassen verwendet. Dabei möchte man oft dafür sorgen, dass alle diese Klassen die gleiche und einzige Instanz des Objekts verwenden – im Programmier-Sprech ein Singleton.
Dabei stellen sich also mehrere Fragen:

- Wo sollen diese Objekte erstellt werden
- Wann soll das erfolgen
- Wie können andere Klassen dann diese Instanz verwenden.

Die Lösung dazu nennt sich Inversion of Control, kurz IoC. Die Art und Weise wie die Objekte zur Verfügung gestellt werden nennt man Dependency Injection (kurz DI). Dazu braucht man einen sog. Service Container, der die einzelnen Objekte enthält. Das muss man sich wie ein Dictionary vorstellen, bei der die Keys die Typen von Klassen/Interfaces sind und die Values dann die (eindeutigen und einzigen) Instanzen dieses Typs:

Key	Value
`typeof(NorthwindContext)`	`new NorthwindContext()`
`typeof(IMyService)`	`new MySpecialService()`
`typeof(IMyOtherService)`	`new MyMockService()`
…	…

1.1 Vorgehensweise

Die Vorgehensweise ist folgende:

- Man erstellt einen Service Container
- In diesem Service Container registriert man alle Objekte, die über DI zur Verfügung stehen sollen
- Bei WPF registriert man auch jene Objekte, die derartige Instanzen konsumieren wollen
- Im Konstruktor gibt man die gewünschte Klasse einfach als Parameter an und DI sorgt dafür, dass man eine entsprechende Instanz erhält.

Bei Web Applikationen mit .Net Core ist dieser Mechanismus automatisch eingebaut, für WPF muss man diese Vorbereitung selbst programmieren.

1.2 Vorbereitung

Man startet mit einem ganz normalen WPF Projekt und fügt die Datenbank Library als Referenz hinzu.

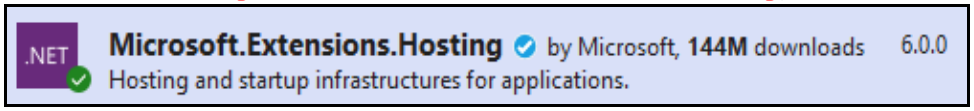
1.2.1 appsettings.json

Wie schon bei Entity Framework DbFirst besprochen, werden Anwendungseinstellungen in appsettings.json konfiguriert. Daher diese Datei jetzt anlegen und den ConnectionString der Datenbank dort eintragen.

Nicht vergessen, diese mit „Copy if newer“ zu markieren.

1.3 NuGet

Dependency Injection wird von .Net Core über das Paket **`Microsoft.Extensions.Hosting`** zur Verfügung gestellt. Diese Assembly muss daher installiert werden (mit NuGet oder in der Package Manger Console mit **`Install-Package Microsoft.Extensions.Hosting`**).



2 Service Container

Folgende Schritte sind notwendig, um den Service Container aufzubauen.

2.1 App.xaml

In App.xaml muss der Eintrag für StartupUri entfernt werden, weil das MainWindow über die registrierte Instanz gestartet wird.
Vorher:

```
<Application x:Class="WpfDependencyInjectionDemo.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfDependencyInjectionDemo"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Nachher:

```
<Application x:Class="WpfDependencyInjectionDemo.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfDependencyInjectionDemo"
             >
    <Application.Resources>

    </Application.Resources>
</Application>
```

2.2 App.xaml.cs

2.2.1 IHost

Zuerst muss ein IHost erzeugt werden, über den der ServiceContainer angesprochen werden kann. Dieser wird über einen Builder erzeugt, bei dem man über ConfigureServices eine Konfigurationsmethode angeben kann.

```
public partial class App : Application
{
  private readonly IHost _host;
  public App()
  {
    _host = Host.CreateDefaultBuilder()
      .ConfigureServices((ctx, services) => ConfigureServices(ctx.Configuration, services))
      .Build();
  }

  private void ConfigureServices(IConfiguration configuration, IServiceCollection services)...
}
```

2.2.2 ConfigureServices

Am verständlichsten ist es aus meiner Sicht, den Service Container in einer eigenen Methode aufzubauen (beim WebBackend mit C# funktioniert es auch so ähnlich).
Diese Methode nennt man üblicherweise **`ConfigureServices`**. In dieser Methode registriert man alle Services, d.h. alle Klassen die man über DI verwenden möchte. Zusätzlich müssen auch die Konsumenten von DI registriert werden, daher auch der Eintrag für MainWindow:

```
internal void ConfigureServices(IConfiguration configuration, IServiceCollection services)
{
  string connectionString = configuration.GetConnectionString("NorthwindDb");
  services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString));
  services.AddSingleton<MainWindow>();
}
```

Hinweise:

- Welcher Datenbank-Provider tatsächlich benutzt wird, ist nicht im Connectionstring enthalten, sondern wird durch den Aufruf **`UseSqlServer`** bestimmt.
- Der Name bei **`GetConnectionString`** ist jener, der in **`appsettings.json`** angegeben wurde.
- IServiceCollection repräsentiert den erwähnten ServiceContainer.

2.2.3 Lebensdauer

Neben **`AddSingleton<>()`** gibt es auch noch **`AddScoped<>()`** und **`AddTransient<>()`**. Die Methode bestimmt die Lebensdauer:

`AddTransient()`	Objekt wird immer neu erzeugt
`AddScoped()`	pro Request wird Objekt neu erzeugt (interessant bei Web-Apps)
`AddSingleton()`	Objekt wird nur ein einziges Mal erzeugt und lebt so lange wie die Applikation

2.2.4 OnStartup

In der überschriebenen Methode OnStartup wird jetzt das MainWindow gestartet (es wurde ja vorher im XAML-File aus StartupUri entfernt). Dazu muss man sich vom ServiceContainer das entsprechende Objekt holen. Den ServiceContainer erhält man über die Property **`Services`** des **`IHost`**, Die Methode dazu heißt dann **`GetRequiredService`**, die generisch ist – der generische Typ bestimmt die Klasse, die man benötigt. In unserem Fall also MainWindow:

```
protected override async void OnStartup(StartupEventArgs e)
{
  await _host.StartAsync();
  var mainWindow = _host.Services.GetRequiredService<MainWindow>();
  mainWindow.Show();
  base.OnStartup(e);
}
```

2.2.5 OnExit

Zu guter Letzt wird beim Beenden der App der Host wieder gestoppt.

```
protected override async void OnExit(ExitEventArgs e)
{
  using (_host)
  {
    await _host.StopAsync();
  }
  base.OnExit(e);
}
```

2.3 MainWindow.cs

Die in ConfigureServices erzeugte Instanz eines registrierten Services (Datenbank,…) muss jetzt einfach nur noch im Konstruktor als Parameter angegeben werden. Das Framework sorgt dafür, dass die zugehörige Instanz zur Verfügung gestellt, also „injected“ wird.

2.3.1 Konstruktor

Im Konstruktor also den Datenbank-Context in der Parameterliste angeben und in einer Instanzvariable speichern:

```
public partial class MainWindow : Window
{
  private readonly NorthwindContext _db;

  public MainWindow(NorthwindContext db)
  {
    InitializeComponent();
    _db = db;
  }

  private void Window_Loaded(object sender, RoutedEventArgs e)...
}
```

2.3.2 Verwendung

In der Verwendung gibt es dann keinen Unterschied. Dieser Code ist also unverändert:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  try
  {
    int nr = _db.Categories.Count();
    Title = $"{nr} Categories";
  }
  catch (Exception ex)
  {
    Title = ex.Message;
  }
}
```

2.4 Logs

In der Konsole werden viele Logs ausgegeben, auch das Hosting betreffen:

```
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\_PR\CSharp\PR4\WpfDependencyInjectionDemo\WpfDependencyInject
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 5.0.0 initialized 'NorthwindContext' using provider 'Microsof
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (27ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT COUNT(*)
      FROM [Categories] AS [c]
```

Falls das stört, kann man das in **`appsettings.json`** anhand von **`LogLevel`**s konfigurieren, z.B.:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting": "Warning",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "ConnectionStrings": {
```

Der Aufbau ist dabei hierarchisch: spezifischere Einstellungen überschreiben allgemeine. So wird in obigem Beispiel alles von Microsoft mit Level „Warning“ geloggt, Microsoft.EntityFrameworkCore jedoch mit Level „Information“.
Die Ausgabe ist jetzt folgende:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 5.0.0 initialized 'NorthwindContext' using provider 'Microsof
' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (29ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT COUNT(*)
      FROM [Categories] AS [c]
```

2.4.1 LogLevel

Die Levels lauten (https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-6.0#log-level):

LogLevel	Value	Method	Description
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should *not* be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category should not write any messages.

3 Erweiterungen

3.1 Weitere Windows

Um auch in allen weiteren Windows Dependency Injection nutzen zu können muss man

- Die Klasse in ConfigureServices registrieren
- Das Window über den ServiceProvider starten
- Daher braucht man dort, wo das Fenster gestartet werden soll, den oben erwähnten ServiceProvider

3.1.1 Registrieren

Das erfolgt (fast) genau wie auch bei MainWindow:

```
private void ConfigureServices(IConfiguration configuration, IServiceCollection services)
{
  string connectionString = configuration.GetConnectionString("NorthwindDb");
  services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString));
  services.AddSingleton<MainWindow>();
  services.AddTransient<OtherWindow>();
}
```

Der Unterschied ist, dass die Registrierung mit AddTransient erfolgt. Dadurch wird das Objekt bei jeder Anforderung neu erzeugt (es ist also genau kein Singleton).

3.1.2 Starten

Das Starten erfolgt über den ServiceProvider, daher muss man sich diesen besorgen. Auch das funktioniert über Dependency Injection:

```
private readonly NorthwindContext _db;
private readonly IServiceProvider _serviceProvider;

public MainWindow(NorthwindContext db, IServiceProvider serviceProvider)
{
  InitializeComponent();
  _db = db;
  _serviceProvider = serviceProvider;
}
```

Damit kann man dann das Window auf dieselbe Weise starten, wie schon MainWindow in App.xaml.cs:

```
private void OpenOtherWindowClick(object sender, RoutedEventArgs e)
{
  var window = _serviceProvider.GetService<OtherWindow>() as Window;
  window?.Show();
}
```

Um den Unterschied von AddTransient zu verstehen Folgendes ausprobieren: mit **`AddSingleton<OtherWindow>`** registrieren und Fenster öffnen, schließen und wieder öffnen.

3.1.3 Dependency Injection

In diesem Window sind dann natürlich alle Objekte des Service Containers verfügbar, indem man sie wieder einfach im Konstruktor anfordert:

```
public OtherWindow(NorthwindContext db)
{
  InitializeComponent();
  _db = db;
  Console.WriteLine($"Db in OtherWindow: {_db.Products.Count()} products");
}
```

3.2 Window mit ViewModel

Für ein ViewModel gilt es analog. Dieses fordert einfach den DbContext an, das verwendende Window gibt seinerseits dann das ViewModel in den Dependencies im Konstruktor an.

3.2.1 ViewModel

Hier ändert sich eigentlich nichts.

```
public class NorthwindViewModel : ObservableObject
{
  public NorthwindViewModel(NorthwindContext db) => NrProducts = db.Products.Count();
  private int _nrProducts;
  public int NrProducts...
}
```

Hinweis: Man kann sich jetzt den Default-Konstruktor sparen, wenn den DataContext als Attribut d:DataContext im Window-Tag angibt und dabei IsDesignTimeCreatable=True nicht angibt:

```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility
xmlns:local="clr-namespace:WpfDependencyInjectionDemo"
d:DataContext="{d:DesignInstance local:NorthwindViewModel}"
mc:Ignorable="d"
```

3.2.2 Window

Hier wird das ViewModel im Konstruktor über DI angefordert:

```
public partial class NorthwindWindow : Window
{
  public NorthwindWindow(NorthwindViewModel viewModel)
  {
    InitializeComponent();
    DataContext = viewModel;
  }
}
```

Das Window muss bei der Anzeige über den ServiceProvider „erzeugt“ werden:

```
serviceProvider.GetRequiredService<NorthwindWindow>().Show();
```

Zum Testen ein Label auf NrProducts des ViewModel binden:

```
<Grid>
  <Label Content="{Binding NrProducts}" HorizontalAlignment="L
</Grid>
```

3.2.3 Registrierung

Man darf dann in ConfigureServices nicht vergessen, sowohl das ViewModel als auch das Window zu registrieren:

```
services.AddTransient<NorthwindWindow>();
services.AddSingleton<NorthwindViewModel>();
```

3.3 Benutzereinstellungen

Für Einstellungen, die vom Endbenutzer verändert werden können, ist die Datei appsettings.json vorgesehen, in der ja schon der ConnectionString für die Datenbank gespeichert wurde.
Alle weiteren Einstellungen werden üblicherweise den einzelnen Klassen ebenfalls wieder über DI zur Verfügung gestellt. Dazu wird empfohlen, das IOptions-Pattern zu implementieren.

3.3.1 appsettings.json

Zuerst die gewünschten Einstellungen in appsettings.json notieren. Dazu kann man sogenannte Sections vergeben, in unserem Fall soll sie MySettings heißen:

```
{
  "Logging": ...,
  "ConnectionStrings": ...,
  "MySettings": {
    "StrVal": "Quaxi",
    "IntVal": 666,
    "UseSomething": true
  }
}
```

3.3.2 Klasse

Jetzt erzeugt man eine Klasse, die üblicherweise genauso heißt wie die Section und jedenfalls Properties enthält, die genauso heißen wie die Keys in den Settings:

```
public class MySettings
{
  public string StrVal { get; set; } = "";
  public int IntVal { get; set; }
  public bool UseSomething { get; set; }
}
```

3.3.3 Zuweisen

In **`ConfigureServices`** kann man über die Methode **`Configure<>()`** jetzt die Section von appsettings.json auf eine Instanz der zugehörigen Klasse mappen:

```
private void ConfigureServices(IConfiguration configuration, IServiceCollection services)
{
  string connectionString = configuration.GetConnectionString("NorthwindDb");
  services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString));
  services.AddSingleton<MainWindow>();
  services.AddSingleton<OtherWindow>();

  services.Configure<MySettings>(configuration.GetSection("MySettings"));
}
```

Klasse

Section

3.3.4 Verwenden

Die Verwendung erfolgt dann wieder über Dependency Injection wobei ein **`IOptions<>`**-Objekt „injected“ wird, deren **`Value`** Property dann die Instanz enthält:

```
private readonly NorthwindContext _db;
private readonly IServiceProvider _serviceProvider;
private readonly MySettings _settings;

public MainWindow(
  NorthwindContext db,
  IServiceProvider serviceProvider,
  IOptions<MySettings> settings)
{
  InitializeComponent();
  _db = db;
  _serviceProvider = serviceProvider;
  _settings = settings.Value;
}
```

Ab dann läuft alles wie erwartet:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  Console.WriteLine($"StrVal = {_settings.StrVal}");
```

```
StrVal = Quaxi
```

3.4 Andere Objekte

Auch beliebige andere Objekte können über DI zur Verfügung gestellt werden. Derartige Objekte nennt man dann üblicherweise Services.

3.4.1 Service erstellen

Das ist eine ganz normale Klasse (die nicht weiß, dass sie mit DI benutzt wird):

```
public class MyService
{
  0 references
  public void DoSomething(string someParameter)
  {
    Console.WriteLine($"MyService::DoSomething with {someParameter}");
  }
}
```

Hinweis: am besten die Services in einem eigenen Verzeichnis erstellen.

3.4.2 Registrieren

Auch Services werden genauso wie andere Klassen (oder Window-Klassen) registriert:

```
private void ConfigureServices(IConfiguration configuration, IServiceCollection services)
{
  string connectionString = configuration.GetConnectionString("NorthwindDb");
  services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString));
  services.AddSingleton<MainWindow>();
  services.AddSingleton<OtherWindow>();

  services.AddSingleton<MyService>();

  services.Configure<MySettings>(configuration.GetSection("MySettings"));
}
```

3.4.3 Verwenden

Dort wo benötigt, lässt man sich die registrierte Instanz im Konstruktor „injecten“:

```
private readonly MyService _myService;
private readonly MySettings _settings;

public MainWindow(
    NorthwindContext db,
    IServiceProvider serviceProvider,
    IOptions<MySettings> settings,
    MyService myService
  )
{
  InitializeComponent();
  _db = db;
  _serviceProvider = serviceProvider;
  _myService = myService;
  _settings = settings.Value;
}
```

Ab dann ist das Objekt überall im Code verfügbar:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  Console.WriteLine($"StrVal = {_settings.StrVal}");
  _myService.DoSomething("alsdkjkl");
  try...
}
```

3.5 Best Practice – Interface

Üblicherweise erzeugt man ein Interface und registriert für dieses Interface eine Instanz einer konkreten Klasse. In der Applikation wird dann überall nur noch mit dem Interface gearbeitet.

3.5.1 Interface

```
public interface IMyOtherService
{
  void Doit();
}
```

3.5.2 Konkreter Typ

```
public class MyOtherServiceImpl : IMyOtherService
{
  public void Doit() => Console.WriteLine("MyOtherServiceImpl::Doit");
}
```

3.5.3 Registrierung

Beim Registrieren gibt man jetzt zwei generische Typen an:

- Zuerst das Interface – das ist der Typ, der von anderen Klassen angefordert werden kann
- Dann die konkrete Klasse – das ist die Instanz, die die anderen Klassen tatsächlich bekommen

```
services.AddSingleton<MyService>();
services.AddSingleton<IMyOtherService, MyOtherServiceImpl>();
```

3.5.4 Verwendung

Im Konstruktor wird jetzt das Interface angegeben – das ist ja der Typ, der im ServiceProvider als „Key“ registriert wurde.

```
    IOptions<MySettings> settings,
    MyService myService,
    IMyOtherService myOtherService
  )
{
  InitializeComponent();
  _db = db;
  _serviceProvider = serviceProvider;
  _myService = myService;
  _myOtherService = myOtherService;
  _settings = settings.Value;
}
```

Wichtig: die verwendende Klasse kennt den tatsächlichen Typ nicht, sie arbeitet nur mit der Abstraktion (=dem Interface). Das wird bei UnitTests interessant, wo man da z.B. für ein Interface eine ganz andere Implementierung injecten kann, ohne in der Klasse selbst auch nur eine einzige Zeile Code ändern zu müssen.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  Console.WriteLine($"StrVal = {_settings.StrVal}");
  _myService.DoSomething("alsdkjkl");
  _myOtherService.Doit();
  try...
}
```

```
StrVal = Quaxi
MyService::DoSomething with alsdkjkl
MyOtherServiceImpl::Doit
```