Injektáž závislostí v .NET

Článek • 28. 11. 2022 • 12 min ke čtení

.NET podporuje vzor návrhu softwaru pro vkládání závislostí (DI), což je technika pro dosažení inverze řízení (IoC) mezi třídami a jejich závislostmi. Injektáž závislostí v .NET je integrovanou součástí architektury spolu s konfigurací, protokolováním a vzorem možností.

Závislost je objekt, na který závisí jiný objekt. Prozkoumejte následující `MessageWriter` třídu pomocí `Write` metody, na které závisí jiné třídy:

C#

```
public class MessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

Třída může vytvořit instanci `MessageWriter` třídy, aby se využila její `Write` metoda. V následujícím příkladu `MessageWriter` je třída závislostí `Worker` třídy:

C#

```
public class Worker : BackgroundService
{
    private readonly MessageWriter _messageWriter = new MessageWriter();

    protected override async Task ExecuteAsync(CancellationToken stoppingTo-
ken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Třída vytvoří a přímo závisí na `MessageWriter` třídě. Pevně zakódované závislosti, například v předchozím příkladu, jsou problematické a měli byste se jim vyhnout z následujících

důvodů:

- Chcete-li nahradit `MessageWriter` jinou implementací `Worker` , musí být třída změněna.
- Pokud `MessageWriter` obsahuje závislosti, musí být také nakonfigurovány `Worker` třídou. Ve velkém projektu s více třídami v závislosti na tom `MessageWriter` se konfigurační kód v aplikaci rozsadí.
- Tato implementace se obtížně testuje. Aplikace by měla používat třídu napodobení nebo zástupných kódů `MessageWriter` , což u tohoto přístupu není možné.

Injektáž závislostí řeší tyto problémy prostřednictvím:

- Použití rozhraní nebo základní třídy k abstrakci implementace závislostí.
- Registrace závislosti v kontejneru služby .NET poskytuje integrovaný kontejner IServiceProviderslužby . Služby se obvykle registrují při spuštění aplikace a připojují se k objektu IServiceCollection. Po přidání všech služeb vytvoříte BuildServiceProvider kontejner služby.
- *Injektáž* služby do konstruktoru třídy, ve které se používá. Architektura přebírá odpovědnost za vytvoření instance závislosti a její likvidaci, když už není potřeba.

Rozhraní například `IMessageWriter` definuje metodu `Write` :

C#

```
namespace DependencyInjection.Example;

public interface IMessageWriter
{
    void Write(string message);
}
```

Toto rozhraní je implementováno konkrétním typem `MessageWriter`:

C#

```
namespace DependencyInjection.Example;

public class MessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");


    }
}
```

```
}
```

Ukázkový kód zaregistruje `IMessageWriter` službu s konkrétním typem `MessageWriter`. Metoda AddScoped zaregistruje službu s vymezenou životností, životností jednoho požadavku. Životnost služeb je popsaná dále v tomto článku.

C#

```
using DependencyInjection.Example;

var builder = Host.CreateDefaultBuilder(args);

builder.ConfigureServices(
    services =>
        services.AddHostedService<Worker>()
            .AddScoped<IMessageWriter, MessageWriter>());

using var host = builder.Build();

host.Run();
```

V předchozím kódu ukázková aplikace:

- Vytvoří instanci tvůrce hostitelů.
- Nakonfiguruje služby registrací:
  - Jako `Worker` hostovaná služba. Další informace najdete v tématu Služby pracovních procesů v .NET.
  - Rozhraní `IMessageWriter` jako služba s vymezeným oborem s odpovídající implementací `MessageWriter` třídy.
- Vytvoří hostitele a spustí ho.

Hostitel obsahuje zprostředkovatele služby injektáže závislostí. Obsahuje také všechny ostatní relevantní služby potřebné k automatickému `Worker` vytvoření instance a poskytnutí odpovídající `IMessageWriter` implementace jako argumentu.

C#

```
namespace DependencyInjection.Example;

public sealed class Worker : BackgroundService
{
    private readonly IMessageWriter _messageWriter;
```

```
    public Worker(IMessageWriter messageWriter) =>
        _messageWriter = messageWriter;

    protected override async Task ExecuteAsync(CancellationToken stoppingTo-
ken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Pomocí modelu DI služba pracovního procesu:

- Nepoužívá konkrétní typ `MessageWriter`, pouze rozhraní, které ho `IMessageWriter` implementuje. To usnadňuje změnu implementace, kterou služba pracovního procesu používá, aniž by se změnila služba pracovního procesu.
- Nevytvoří instanci objektu `MessageWriter`. Instance je vytvořena kontejnerem DI.

Implementaci `IMessageWriter` rozhraní lze vylepšit pomocí integrovaného rozhraní API protokolování:

C#

```
namespace DependencyInjection.Example;

public class LoggingMessageWriter : IMessageWriter
{
    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation("Info: {Msg}", message);
}
```

Aktualizovaná `ConfigureServices` metoda zaregistruje novou `IMessageWriter` implementaci:

C#

```
static IHostBuilder CreateHostBuilder(string[] args) =>
```

```
static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((_, services) =>
            services.AddHostedService<Worker>()
                    .AddScoped<IMessageWriter, LoggingMessageWriter>());
```

Metoda `CreateHostBuilder` používá typy `IHostBuilder` a `Host`. Aby je bylo možné použít, musí se odkazovat na balíčky `Microsoft.Extensions.DependencyInjection` a `Microsoft.Extensions.Hosting`.

`LoggingMessageWriter` závisí na ILogger<TCategoryName>, který požaduje v konstruktoru. `ILogger<TCategoryName>` je služba poskytovaná architekturou.

Není neobvyklé používat injektáž závislostí zřetězeným způsobem. Každá požadovaná závislost si pak vyžádá své vlastní závislosti. Kontejner přeloží závislosti v grafu a vrátí plně vyřešenou službu. Souhrnná sada závislostí, které je třeba vyřešit, se obvykle označuje jako *strom závislostí*, *graf závislostí* nebo *graf objektů*.

Kontejner se vyřeší `ILogger<TCategoryName>` tak, že využívá (obecné) otevřené typy, čímž eliminuje nutnost registrace každého (obecného) vytvořeného typu.

S terminologií injektáže závislostí služba:

- Obvykle je objekt, který poskytuje službu jiným objektům, jako `IMessageWriter` je služba.
- Nesouvisí s webovou službou, i když služba může používat webovou službu.

Architektura poskytuje robustní systém protokolování. Implementace `IMessageWriter` uvedené v předchozích příkladech byly napsány tak, aby demonstrovaly základní DI, nikoli k implementaci protokolování. Většina aplikací by neměla psát protokolovací nástroje. Následující kód ukazuje použití výchozího protokolování, které vyžaduje `Worker` pouze registraci jako `ConfigureServices` hostované služby AddHostedService:

C#

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger) =>
        _logger = logger;

    protected override async Task ExecuteAsync(CancellationToken stoppingTo-
ken)
```

```
ken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Pomocí předchozího kódu není potřeba aktualizovat `ConfigureServices`, protože protokolování poskytuje architektura.

Pravidla zjišťování více konstruktorů

Když typ definuje více než jeden konstruktor, má zprostředkovatel služby logiku pro určení, který konstruktor se má použít. Je vybrán konstruktor s většinou parametrů, u kterých jsou typy di-přeložitelné. Představte si následující ukázkovou službu jazyka C#:

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(FooService fooService, BarService barService)
    {
        // omitted for brevity
    }
}
```

V předchozím kódu předpokládejme, že bylo přidáno protokolování, které je možné přeložit od poskytovatele služby, ale `FooService` typy a `BarService` nejsou. Konstruktor s parametrem `ILogger<ExampleService>` slouží k překladu `ExampleService` instance. I když

existuje konstruktor, který definuje více parametrů, `FooService` typy a `BarService` nejsou

di-přeložitelné.

Pokud při zjišťování konstruktorů existuje nejednoznačnost, vyvolá se výjimka. Představte si následující ukázkovou službu jazyka C#:

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(IOptions<ExampleOptions> options)
    {
        // omitted for brevity
    }
}
```

> ⚠ **Upozornění**
>
> Kód `ExampleService` s nejednoznačnými parametry typu DI by vyvolal výjimku. Neuděláte to – má to ukázat, co znamená "nejednoznačné typy s možností překladu DI".

V předchozím příkladu jsou tři konstruktory. První konstruktor je bez parametrů a nevyžaduje žádné služby od poskytovatele služeb. Předpokládejme, že protokolování a možnosti byly přidány do kontejneru DI a jsou službami, které lze přeložit. Když se kontejner DI pokusí typ přeložit `ExampleService` , vyvolá výjimku, protože dva konstruktory jsou nejednoznačné.

Nejednoznačnosti se můžete vyhnout definováním konstruktoru, který místo toho přijímá oba typy s možností překladu DI:

C#

```
public class ExampleService
```

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(
        ILogger<ExampleService> logger,
        IOptions<ExampleOptions> options)
    {
        // omitted for brevity
    }
}
```

Registrace skupin služeb pomocí rozšiřujících metod

Rozšíření společnosti Microsoft používají konvenci pro registraci skupiny souvisejících služeb. Konvencí je použití jedné `Add{GROUP_NAME}` rozšiřující metody k registraci všech služeb požadovaných funkcí architektury. Metoda rozšíření například AddOptions registruje všechny služby potřebné pro použití možností.

Služby poskytované architekturou

Metoda `ConfigureServices` registruje služby, které aplikace používá, včetně funkcí platformy. Na začátku má zadaný objekt `ConfigureServices` služby definované architekturou v závislosti na tom, `IServiceCollection` jak byl hostitel nakonfigurován. Pro aplikace založené na šablonách .NET rozhraní registruje stovky služeb.

Následující tabulka uvádí malou ukázku těchto služeb registrovaných v architektuře:

Typ služby	Doba platnosti
Microsoft.Extensions.DependencyInjection.IServiceScopeFactory	Singleton
IHostApplicationLifetime	Singleton
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton

Typ služby	Doba platnosti

Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Přechodná
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticListener	Singleton
System.Diagnostics.DiagnosticSource	Singleton

Životnost služeb

Služby je možné zaregistrovat v některé z následujících životností:

- Přechodná
- Rozsahem
- Singleton

Následující části popisují každou z předchozích životností. Zvolte vhodnou dobu životnosti pro každou registrovanou službu.

Přechodná

Služby přechodné životnosti se vytvářejí pokaždé, když jsou požadovány z kontejneru služby. Tato životnost je nejvhodnější pro jednoduché bezstavové služby. Registrace přechodných služeb pomocí AddTransient.

V aplikacích, které zpracovávají požadavky, se na konci žádosti odstraní přechodné služby.

Rozsahem

U webových aplikací vymezená doba života označuje, že služby se vytvářejí jednou pro každý požadavek klienta (připojení). Registrace služeb s vymezeným oborem pomocí AddScoped.

V aplikacích, které zpracovávají požadavky, se na konci žádosti odstraní vymezené služby.

Při použití Entity Framework Core AddDbContext metoda rozšíření ve výchozím nastavení registruje `DbContext` typy s vymezenou životností.

> [!NOTE]
> **Poznámka**
>
> ***Nevyřešujte*** službu s vymezeným oborem z jednoúčelové služby a dávejte pozor, abyste to neudělali nepřímo, například prostřednictvím přechodné služby. Může to způsobit nesprávný stav služby při zpracování následných požadavků. Je v pořádku:
>
> - Řešení potíží s jednoúčelovou službou z vymezené nebo přechodné služby
> - Řešení problému s vymezenou službou z jiné služby s vymezeným oborem nebo přechodnou službou

Ve výchozím nastavení ve vývojovém prostředí vyvolá překlad služby z jiné služby s delší životností výjimku. Další informace najdete v tématu Ověření oboru.

Singleton

Jednoúčelové služby se vytvářejí takto:

- Při prvním vyžádání.
- Vývojář při poskytování instance implementace přímo do kontejneru. Tento přístup je zřídka nutný.

Každý další požadavek implementace služby z kontejneru injektáže závislostí používá stejnou instanci. Pokud aplikace vyžaduje jednoúčelové chování, povolte kontejneru služby spravovat životnost služby. Neimplementujte vzor jednoúčelového návrhu a zadávejte kód pro odstranění jednoduchého objektu. Služby by nikdy neměly být odstraněny kódem, který přeložil službu z kontejneru. Pokud je typ nebo továrna registrována jako singleton, kontejner automaticky odstraní singleton.

Zaregistrujte jednoúčelové služby pomocí AddSingleton. Jednoúčelové služby musí být bezpečné pro přístup z více vláken a často se používají v bezstavových službách.

V aplikacích, které zpracovávají požadavky, se ServiceProvider při vypínání aplikace odstraní jednoúčelové služby. Vzhledem k tomu, že paměť se neuvolní, dokud se aplikace nevypnou, zvažte použití paměti s jednou službou.

Metody registrace služby

Architektura poskytuje metody rozšíření registrace služeb, které jsou užitečné v konkrétních

scénářích:

Metoda	Automaticky object K dispozici	Několik implementace	Předávání argumentů

`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()`
 	
Příklad:
 	
`services.AddSingleton<IMyDep, MyDep>();`
	Ano	Ano	Ne

`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})`
 	
Příklady:
 	
`services.AddSingleton<IMyDep>(sp => new MyDep());`
`services.AddSingleton<IMyDep>(sp => new MyDep(99));`
	Ano	Ano	Ano

`Add{LIFETIME}<{IMPLEMENTATION}>()`
 	
Příklad:
 	
`services.AddSingleton<MyDep>();`
	Ano	Ne	Ne

`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})`
 	
Příklady:
 	
`services.AddSingleton<IMyDep>(new MyDep());`
`services.AddSingleton<IMyDep>(new MyDep(99));`
	Ne	Ano	Ano

`AddSingleton(new {IMPLEMENTATION})` Příklady:	Ne	Ne	Ano
Metoda	**Automaticky**	**Několik**	**Předávání**

	object	implementace	argumentů
	K dispozici		

```
services.AddSingleton(new MyDep());
services.AddSingleton(new MyDep(99));
```

Další informace o likvidaci typů najdete v části Likvidace služeb .

Registrace služby pouze s typem implementace je ekvivalentní registraci této služby se stejnou implementací a typem služby. To je důvod, proč nelze zaregistrovat více implementací služby pomocí metod, které nemají explicitní typ služby. Tyto metody můžou registrovat více *instancí* služby, ale všechny budou mít stejný typ *implementace* .

Libovolnou z výše uvedených metod registrace služby lze použít k registraci více instancí služby stejného typu služby. V následujícím příkladu `AddSingleton` je volána dvakrát s typem `IMessageWriter` služby. Druhé volání přepíše `AddSingleton` předchozí volání, pokud je vyřešeno jako `IMessageWriter` , a přidá se k předchozímu volání, pokud je vyřešeno více služeb prostřednictvím `IEnumerable<IMessageWriter>` . Služby se zobrazují v pořadí, v jakém byly zaregistrovány, když byly vyřešeny přes `IEnumerable<{SERVICE}>` .

C#

```
using ConsoleDI.IEnumerableExample;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

using IHost host = CreateHostBuilder(args).Build();

_ = host.Services.GetService<ExampleService>();

await host.RunAsync();

static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((_, services) =>
            services.AddSingleton<IMessageWriter, ConsoleMessageWriter>()
                    .AddSingleton<IMessageWriter, LoggingMessageWriter>()
                    .AddSingleton<ExampleService>());
```

Předchozí ukázkový zdrojový kód registruje dvě implementace objektu `IMessageWriter` .

C#

```
using System.Diagnostics;
```

```
using System.Diagnostics;

namespace ConsoleDI.IEnumerableExample;

public sealed class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is LoggingMessageWriter);

        var dependencyArray = messageWriters.ToArray();
        Trace.Assert(dependencyArray[0] is ConsoleMessageWriter);
        Trace.Assert(dependencyArray[1] is LoggingMessageWriter);
    }
}
```

Definuje `ExampleService` dva parametry konstruktoru: jeden `IMessageWriter` a `IEnumerable<IMessageWriter>`. Jediný `IMessageWriter` je poslední zaregistrovanou implementací, zatímco představuje `IEnumerable<IMessageWriter>` všechny registrované implementace.

Architektura také poskytuje `TryAdd{LIFETIME}` rozšiřující metody, které službu registrují pouze v případě, že ještě není zaregistrovaná implementace.

V následujícím příkladu se volání registruje `AddSingleton ConsoleMessageWriter` jako implementace pro `IMessageWriter`. Volání `TryAddSingleton` nemá žádný účinek, protože `IMessageWriter` už má zaregistrovanou implementaci:

C#

```
services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
services.TryAddSingleton<IMessageWriter, LoggingMessageWriter>();
```

Nemá `TryAddSingleton` žádný vliv, protože už byl přidán a pokus se nezdaří. Příkaz `ExampleService` by tvrdil následující:

C#

```
public class ExampleService
{
    public ExampleService(

        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
```

```
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is ConsoleMessageWriter);
        Trace.Assert(messageWriters.Single() is ConsoleMessageWriter);
    }
}
```

Další informace naleznete v tématu:

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

Metody TryAddEnumerable(ServiceDescriptor) zaregistrují službu pouze v případě, že ještě neexistuje implementace *stejného typu*. Více služeb se řeší přes `IEnumerable<{SERVICE}>`. Při registraci služeb přidejte instanci, pokud ještě nebyl přidán jeden ze stejných typů. Autoři knihovny používají `TryAddEnumerable`, aby se zabránilo registraci více kopií implementace v kontejneru.

V následujícím příkladu se první volání `TryAddEnumerable` registruje `MessageWriter` jako implementace pro `IMessageWriter1`. Druhé volání se zaregistruje `MessageWriter` pro `IMessageWriter2`. Třetí volání nemá žádný účinek, protože `IMessageWriter1` již má zaregistrovanou implementaci `MessageWriter`:

C#

```
public interface IMessageWriter1 { }
public interface IMessageWriter2 { }

public class MessageWriter : IMessageWriter1, IMessageWriter2
{
}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,
MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter2,
MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,
MessageWriter>());
```

Registrace služby je obecně nezávislá na objednávce s výjimkou registrace více

implementací stejného typu.

`IServiceCollection` je kolekce ServiceDescriptor objektů. Následující příklad ukazuje, jak zaregistrovat službu vytvořením a přidáním `ServiceDescriptor`:

C#

```
string secretKey = Configuration["SecretKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMessageWriter),
    _ => new DefaultMessageWriter(secretKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

Předdefinované `Add{LIFETIME}` metody používají stejný přístup. Podívejte se například na zdrojový kód AddScoped .

Chování injektáže konstruktoru

Služby je možné vyřešit pomocí:

- IServiceProvider
- ActivatorUtilities:
  - Vytvoří objekty, které nejsou zaregistrované v kontejneru.
  - Používá se s některými funkcemi architektury.

Konstruktory mohou přijímat argumenty, které nejsou poskytovány injektací závislostí, ale argumenty musí přiřadit výchozí hodnoty.

Pokud jsou služby vyřešeny nástrojem `IServiceProvider` nebo `ActivatorUtilities`, injektáž konstruktoru vyžaduje *veřejný* konstruktor.

Když jsou služby vyřešeny nástrojem `ActivatorUtilities`, injektáž konstruktoru vyžaduje, aby existoval pouze jeden použitelný konstruktor. Přetížení konstruktoru jsou podporována, ale může existovat pouze jedno přetížení, jehož argumenty mohou být splněny prostřednictvím injektáže závislostí.

Ověření oboru

Když aplikace běží v `Development` prostředí a volá Metodu CreateDefaultBuilder k sestavení

hostitele, výchozí poskytovatel služeb provede kontroly, aby ověřil, že:

- Služby s vymezeným oborem se od poskytovatele kořenových služeb nepřeloží.
- Služby s vymezeným oborem se nevkládají do jednoúčelových objektů.

Poskytovatel kořenové služby se vytvoří při BuildServiceProvider zavolání. Životnost poskytovatele kořenové služby odpovídá životnosti aplikace při spuštění aplikace a při vypnutí aplikace se odstraní.

Služby s vymezeným oborem jsou uvolněny kontejnerem, který je vytvořil. Pokud se služba s vymezeným oborem vytvoří v kořenovém kontejneru, doba života služby se ve skutečnosti zvýší na singleton, protože kořenový kontejner ji vyhodí jenom při vypnutí aplikace. Ověřování rozsahů služby zachycuje tyto situace při `BuildServiceProvider` zavolání.

Oborové scénáře

Je IServiceScopeFactory vždy registrován jako singleton, ale IServiceProvider může se lišit v závislosti na životnosti třídy obsahující. Pokud například přeložíte služby z oboru a kterákoli z těchto služeb převezme IServiceProvider, bude se jednat o instanci s vymezeným oborem.

Chcete-li dosáhnout vymezení rozsahu služeb v rámci implementace IHostedService, jako BackgroundServiceje například , *nevkládat* závislosti služby prostřednictvím injektáže konstruktoru. Místo toho vložte IServiceScopeFactory, vytvořte obor a pak vyřešte závislosti z oboru tak, aby se používala odpovídající životnost služby.

C#

```
namespace WorkerScope.Example;

public sealed class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IServiceScopeFactory _serviceScopeFactory;

    public Worker(ILogger<Worker> logger, IServiceScopeFactory serviceScope-
Factory) =>
        (_logger, _serviceScopeFactory) = (logger, serviceScopeFactory);

    protected override async Task ExecuteAsync(CancellationToken stoppingTo-
ken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
```

```
        {
            using (IServiceScope scope = _serviceScopeFactory.CreateScope())
            {
                try
                {
                    _logger.LogInformation(
                        "Starting scoped work, provider hash: {hash}.",
                        scope.ServiceProvider.GetHashCode());

                    var store =
scope.ServiceProvider.GetRequiredService<IObjectStore>();
                    var next = await store.GetNextAsync();
                    _logger.LogInformation("{next}", next);

                    var processor =
scope.ServiceProvider.GetRequiredService<IObjectProcessor>();
                    await processor.ProcessAsync(next);
                    _logger.LogInformation("Processing {name}.", next.Name);

                    var relay =
scope.ServiceProvider.GetRequiredService<IObjectRelay>();
                    await relay.RelayAsync(next);
                    _logger.LogInformation("Processed results have been re-
layed.");

                    var marked = await store.MarkAsync(next);
                    _logger.LogInformation("Marked as processed: {next}", mar-
ked);
                }
                finally
                {
                    _logger.LogInformation(
                        "Finished scoped work, provider hash: {hash}.{nl}",
                        scope.ServiceProvider.GetHashCode(),
Environment.NewLine);
                }
            }
        }
    }
}
```

V předchozím kódu, zatímco je aplikace spuštěná, služba na pozadí:

- Závisí na .IServiceScopeFactory
- Vytvoří pro IServiceScope překlad dalších služeb.
- Řeší služby s vymezeným oborem pro využití.

- Funguje při zpracování objektů a jejich následném předávání a nakonec je označí jako

zpracované.

Z ukázkového zdrojového kódu můžete vidět, jak IHostedService implementace nástroje můžou těžit z omezené životnosti služeb.

Viz také

- Použití injektáže závislostí v .NET
- Pokyny pro injektáž závislostí
- Injektáž závislostí v ASP.NET Core
- Vzory konference NDC pro vývoj DI aplikací
- Princip explicitních závislostí
- Inverze kontejnerů řízení a vzor injektáže závislostí (Martin Fowler)
- V úložišti github.com/dotnet/extensions by se měly vytvářet chyby DI.