IoC Контейнеры и DI в .NET

Докладчик: Закарлюка Иван Владимирович

IoC

IoC (Inversion of Control, Инверсия Управления) — архитектурный принцип, который гласит, что зависимость в приложении должна быть направлена в сторону абстракции, а не на детали реализации

DI

DI (Dependency Injection, Внедрение Зависимостей) — метод, реализующий принцип IoC, который позволяет передавать необходимые объектам зависимости извне, а не создавать их внутри самих объектов

IoC и DI в .NET

IoC-контейнер представлен в .NET в библиотеке `Microsoft.Extensions.DependencyInjection`

Пример

```
namespace Example;
public interface IMyService
{
    int Execute();
}

public class MyService : IMyService
{
    public int Execute() {
        return 42;
    }
}
```

Пример

```
namespace Example;
public class DeepThought(IMyService service)
{
    public IMyService service = service;

    public void Answer()
    {
        var answer = service.Execute();
        Console.Write("The Answer to the Ultimate");
        Console.Write("Question of Life, the Universe, ");
        Console.Write($"and Everything is {answer}\n");
    }
}
```

Пример

```
using Example;
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();
services.AddSingleton<IMyService, MyService>();
services.AddSingleton<DeepThought>();
var provider = services.BuildServiceProvider();
var computer = provider.GetService<DeepThought>();
computer.Answer();
```

Способы добавления сервиса — виды Lifetime

- Singleton — создается единожды, в первый раз когда он вызван
- Transient — новый экземпляр создается при каждом вызове
- Scoped — для веб-приложений создается один раз для каждого запроса. В общем случае создается в каждом новом scope

Пример для scoped

```
var services = new ServiceCollection();
services.AddScoped<IMyService, MyService>();
services.AddSingleton<DeepThought>();

var provider = services.BuildServiceProvider();
var computer = provider.GetService<DeepThought>();
computer.Answer();

var scopeFactory = provider
                .GetService<IServiceScopeFactory>();
using (var scope = scopeFactory.CreateScope())
{
    computer = scope.ServiceProvider
                    .GetService<DeepThought>();
    computer.Answer(); // будет создан новый MyService,
    // поскольку находимся в другом scope
}
```

Способы добавления сервиса

- Add{LIFETIME}<{SERVICE},{IMPLEMENTATION}>() — добавляет реализацию для сервиса в указанном lifetime
- Add{LIFETIME}<{IMPLEMENTATION}> — аналогично предыдущему
- Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION}) — позволяет передавать значения в конструктор класса-реализации
- AddSingleton<{SERVICE}>(new {IMPLEMENTATION}) — не удаляет объект-реализацию автоматически при удалении ServiceCollection
- AddSingleton(new {IMPLEMENTATION}) — аналогично второму способу

Способы добавления сервиса

- TryAdd{LIFETIME}<{SERVICE}, {IMPLEMENTATION}> — добавит реализацию, только если к этому сервису ещё не было добавлено ни одной реализации
- TryAddEnumerable(ServiceDescriptor) — позволяет добавить несколько реализаций к одному сервису и игнорирует реализации, которые уже были добавлены
- Add(ServiceDescriptor)

ServiceDescriptor

Через ServiceDescriptor можно описать любой сервис

```
ServiceDescriptor.Describe(
    serviceType: typeof(IMyService),
    implementationType: typeof(MyService),
    lifetime: ServiceLifetime.Singleton)
```

Добавление нескольких реализаций

```
using Example;
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();
services.AddSingleton<IMyService, MyService>();
services.AddSingleton<IMyService, MyWrongService>();
services.AddSingleton<IMyService, MySlightlyWrongService>();
services.AddSingleton<DeepThought>();
var provider = services.BuildServiceProvider();
var computer = provider.GetService<DeepThought>();
computer.Answer(); // The Answer to ... is 43
```

Передается последняя добавленная реализация

Добавление нескольких реализаций

```
public IEnumerable<IMyService> services;

public DeepThought(IEnumerable<IMyService> services) {
    this.services = services;
}

public void Answer() {
    foreach (var service in this.services) {
        var answer = service.Execute();
        Console.Write("The Answer to the Ultimate");
        Console.Write("Question of Life, the Universe, ");
        Console.Write($"and Everything is {answer}\n");
    }
}
```

Консольный вывод

The Answer to the Ultimate Question of Life, the Universe, and Everything is 42
The Answer to the Ultimate Question of Life, the Universe, and Everything is 1
The Answer to the Ultimate Question of Life, the Universe, and Everything is 43

Несколько разных конструкторов

```
public class DeepThought
{
    public DeepThought(IMyService service)
    {
        Console.WriteLine(1);
    }

    public DeepThought(IMyOption option, IMyService service)
    {
        Console.WriteLine(2);
    }
}
```

Консольный вывод: 2

Выбирается конструктор с наибольшим количеством входных параметров

Несколько разных конструкторов

```
public class DeepThought
{
    public DeepThought(IMyService service)
    {
        Console.WriteLine(1);
    }

    public DeepThought(IMyOption optrion)
    {
        Console.WriteLine(2);
    }
}
```

System.InvalidOperationException — контейнер не знает, какой конструктор выбрать

1. Host.CreateDefaultBuilder()
2. Host.CreateApplicationBuilder()
3. WebHost.CreateDefaultBuilder()
4. WebApplication.CreateBuilder()
5. WebAssemblyHostBuilder.CreateDefault
6. MauiApp.CreateBuilder

Создают IServiceCollection и регистрируют сервисы в зависимости от конфигурации, после чего дают возможность регистрировать и другие сервисы

Пример

```
using Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.
        CreateApplicationBuilder(args);
builder.Services.AddSingleton<DeepThought>();
builder.Services.AddSingleton<IMyService, MyService>();

using IHost host = builder.Build();
host.Run();
```

Конфигурация

Библиотека позволяет конфигурировать сервисы. Например, пусть есть

```
public class MyOptions
{
    public string Setting1 { get; set; }
    public string Setting2 { get; set; }
}
```

И файл appsettings.json

```
{
  "MyOptions": {
    "Setting1": "Value42",
    "Setting2": 42
  }
}
```

Конфигурация

```
var services = new ServiceCollection();
var configuration = new ConfigurationBuilder()
    .AddJsonFile(
        "appsettings.json",
        optional: false,
        reloadOnChange: true)
    .Build();
services.Configure<MyOptions>(configuration
        .GetSection("MyOptions"));
services.AddSingleton<MyOptions>();
...
```

Источники

1. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection
2. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-basics
3. https://habr.com/ru/articles/595613/
4. https://learn.microsoft.com/en-us/dotnet/api/