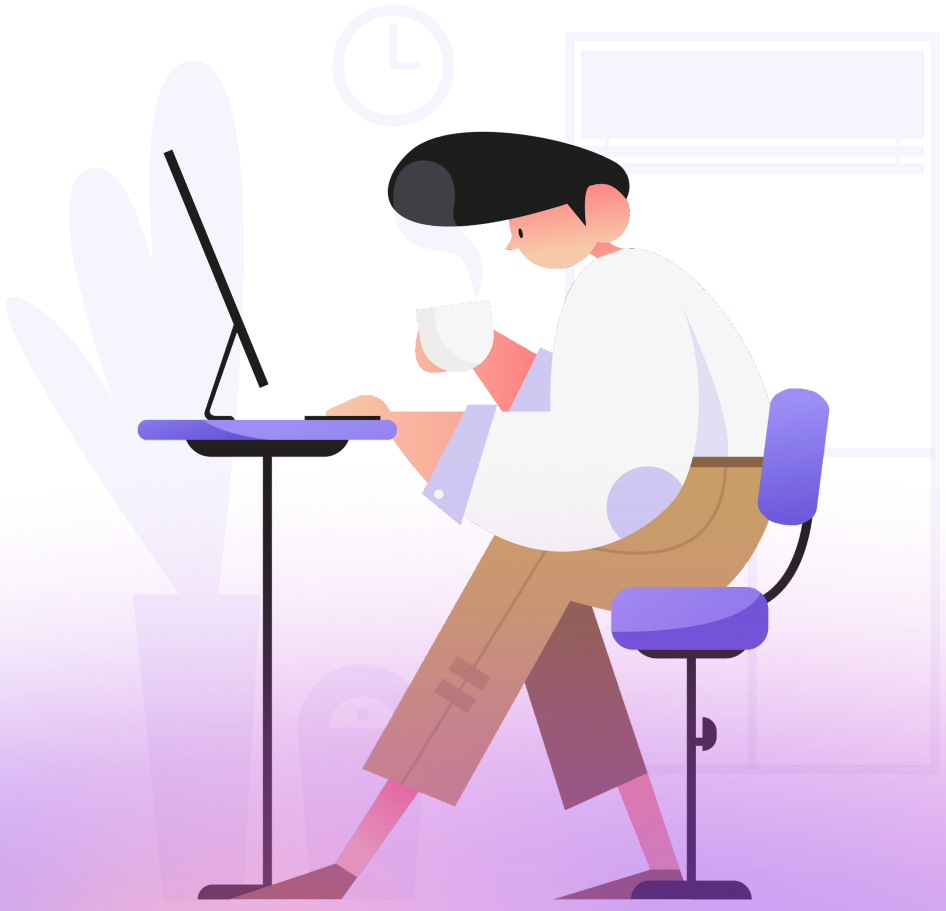
GeekBrains

ASP.NET MVC и углубление в изучение C#

Autofac



На этом уроке

1. Познакомимся с популярным контейнером зависимостей Autofac.
2. Узнаем как им пользоваться в простом виде.
3. Рассмотрим его дополнительный функционал создания паттернов Адаптер и Декоратор.
4. Научимся создавать композиты и агрегаторы.
5. Повторим пройденные материалы по паттернам.

Оглавление

Внедрение зависимостей на базе Autofac

Внедрение зависимостей важнейший элемент разработки программного обеспечения на .net платформе. Как можно уже помнить, благодаря этому подходу можно достичь минимальной связанности кода, что даст большую гибкость и динамичность вашему приложению. На сегодняшний день, почти ни один из проектов на .net не разрабатывается без помощи какого-либо контейнера зависимостей. Вы уже пользовались одним из таких контейнеров, стандартной реализацией от Microsoft – IServiceCollection. Именно этот контейнер используется из коробки в asp net приложениях, что в WebAPI, что и в других.

Но зачастую, функционала IServiceCollection контейнера не всегда хватает. Это объясняет то, что на рынке он относительно не так давно, и следует более суровым правилам, нежели чем старички. И именно о старичках и пойдет речь далее.

Autofac, как один из лидеров на рынке внедрения зависимостей

Контейнер Autofac очень хорошо зарекомендовал себя на рынке из-за функционала и производительности. Он легок в понимании. Хорошая сопроводительная справка и отличный набор функциональности Autofac создает преимущества по сравнению с дефолтным контейнером. Замена на Autofac выглядит заманчивой, но стоит отметить, что у этой замены есть и недостатки – зависимость от функционала, которого не может не быть в других контейнерах, если по какой-либо причине нужно будет заменить и Autofac.

Первое приложение

Создадим консольное приложение, в котором ознакомимся с тем, как работает контейнер Autofac.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Interview.Sample
{
	class Program
	{
	static void  Main(string[] args)
	{

		ISomeService service = new SomeService();

		service.PrintHello();

	}
	}

	public interface ISomeService
	{
	void PrintHello();
	}

	public sealed class SomeService : ISomeService
	{
	public void PrintHello()
	{
		Console.WriteLine($"Hello from {nameof(SomeService)}");
	}
	}
}
```

В данном примере есть контракт некого сервиса ISomeService. Экземпляр данного типа пока создается через оператор new. По факту – ничего нового и сложного. Так как это слишком связанный код, то стоит его по максимуму сделать несвязанным. Для разминки самое то.

Для начала добавим контейнер Autofac через пакетный менеджер nuget. После этого перепишем код, который создает ISomeService. Но для начала стоит вспомнить паттерн Builder. В этом паттерне сначала создается его Builder объект, затем через этот объект настраивается то, как должен создаваться контейнер, а точнее его зависимости и уже после происходит создается сам паттерн.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();

            builder.RegisterType<SomeService>().As<ISomeService>();
            IContainer container  = builder.Build();

            ISomeService service = container.Resolve<ISomeService>();

            service.PrintHello();

      }
      }

      public interface ISomeService
      {
      void PrintHello();
      }

      public sealed class SomeService : ISomeService
      {
      public void PrintHello()
      {
            Console.WriteLine($"Hello from {nameof(SomeService)}");
      }
      }
}
```

Как можно увидеть, максимальные изменения были в только метод Main. Для создания контейнера Autofac сначала создается некий Builder, через который идет конфигурация зависимостей, и уже после – создается сам контейнер.

И в первом случае, и во втором случае на консоль должна быть выведена надпись.

```
Hello from SomeService
```

Это самый простой способ регистрации зависимостей. Если, к примеру, переписать код в таком виде, то каждый раз при запросе контракта из контейнера будет создаваться новый объект.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();

            builder.RegisterType<SomeService>().As<ISomeService>();
            IContainer container  = builder.Build();

            ISomeService service1 = container.Resolve<ISomeService>();
            ISomeService service2 = container.Resolve<ISomeService>();

            if (service1.GetHashCode() == service2.GetHashCode())
            {
                Console.WriteLine("Объекты равны!");

            return;
            }

            Console.WriteLine("Объекты не равны!");


      }
      }

      public interface ISomeService
      {
      void PrintHello();
      }

      public sealed class SomeService : ISomeService
      {
      public void PrintHello()
      {
            Console.WriteLine($"Hello from {nameof(SomeService)}");
      }
      }
}
```

В результате выполнения на консоли появится сообщение, что данные объекты не равны, хотя являются одним и тем же контрактом.

```
Объекты не равны!
```

Регистрация зависимостей – singleton

Пожалуй, синглтон является самой популярной регистрацией зависимостей. Это подразумевает, что один и тот же экземпляр зависимости будет внедряться каждый раз при ее запросе. Конечно же в контексте одного и того же процесса.

Для регистрации зависимости как синглтон стоит использовать метод расширения SingleInstance

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
        {
            var builder = new ContainerBuilder();


builder.RegisterType<SomeService>().As<ISomeService>().SingleInstance();
            IContainer container  = builder.Build();

            ISomeService service1 = container.Resolve<ISomeService>();
            ISomeService service2 = container.Resolve<ISomeService>();

            if (service1.GetHashCode() == service2.GetHashCode())
            {
            Console.WriteLine("Объекты равны!");

            return;
            }

            Console.WriteLine("Объекты не равны!");

      }
      }
```

```
      public interface ISomeService
      {
      void PrintHello();
      }

      public sealed class SomeService : ISomeService
    {
      public void PrintHello()
      {
            Console.WriteLine($"Hello from {nameof(SomeService)}");
      }
      }
}
```

Результатом будет вывод строки, что данные объекты равны. Что и требовалось доказать.

```
Объекты равны!
```

На протяжении большего времени именно этот способ будет одним из самых часто используемых в коде.

Адаптер

Паттерн адаптер был достаточно сильно разобран на предыдущих уроках. Именно ручное его создание. Но, как говорится выше по тексту, Autofac богат на дополнительный функционал. Такие паттерны, как адаптер, декоратор и фабрика встроены в него изначально.

Допустим, стоит задача создать адаптеры Operation на некие контракты IPipelineItem. Самый простой пример того, быстрой адаптации через контейнер Autofac является метод расширения RegisterAdpater.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;
using Autofac.Features.Metadata;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();


builder.RegisterType<FirstPipelineItem>().As<IPipelineItem>().WithMetadata
("Name", "First pipe item");

builder.RegisterType<SecondPipelineItem>().As<IPipelineItem>().WithMetadat
a("Name", "Second pipe item");

            builder.RegisterAdapter<Meta<IPipelineItem>, Operation>(
            cmd => new Operation(cmd.Value,
(string)cmd.Metadata["Name"]));

            IContainer container  = builder.Build();

            IReadOnlyList<Operation> operations =
container.Resolve<IEnumerable<Operation>>().ToList();

            foreach (Operation operation in operations)
            {
                Console.WriteLine($"Operation name is {operation.Name}");
            operation.Execute();
            }

      }
      }

      public interface IPipelineItem
      {
      string Name { get; }

      void Run();
      }

      public abstract class PipelineItem : IPipelineItem
      {
      public abstract string Name { get; }

      public abstract void Run();
      }

      public sealed class FirstPipelineItem : PipelineItem
```

```
      {
      public override string Name => $"{nameof(FirstPipelineItem)}";

      public override void Run()
      {
            Console.WriteLine($"Hello from {Name}");
      }
      }

      public sealed class SecondPipelineItem : PipelineItem
      {
      public override string Name => $"{nameof(SecondPipelineItem)}";

      public override void Run()
      {
            Console.WriteLine($"Hello from {Name}");
      }
      }

      public sealed class Operation
      {
      private readonly IPipelineItem _pipelineItem;
      private readonly string _name;

      public Operation(IPipelineItem pipelineItem, string name)
      {
            _pipelineItem = pipelineItem;
            _name = name;
      }

       public string Name => _name;

      public void Execute()
      {
            _pipelineItem.Run();
      }
      }
}
```

Если внимательно посмотреть на код, то все контракты IPipelineItem были зарегистрированы вместе с дополнительной метаинформацией, которую можно будет использовать впоследствии при передаче в конструктор адаптера Operation, помимо самого IPipelineItem. В данном случае передается название контракта в свободном виде.

Результатом данного кода будет сообщения в консоли.

```
Operation name is First pipe item

Hello from FirstPipelineItem

Operation name is Second pipe item

Hello from SecondPipelineItem
```

Декоратор

Помимо паттерна Адаптер, Autofac поддерживает и паттерн Декоратор, который так же проходили ранее. Возьмем предыдущий пример и слегка его изменим. Класс Operation переименуем в DecorationOperation, унаследуем от IPipelineItem и изменим его с условием нового контракта. После укажем при регистрации, что теперь он является декоратором для всех IPipelineItem-объектов.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;
using Autofac.Features.Metadata;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();

            builder.RegisterType<FirstPipelineItem>().As<IPipelineItem>();

builder.RegisterType<SecondPipelineItem>().As<IPipelineItem>();

            builder.RegisterDecorator<DecoratorOperation,
IPipelineItem>();

            IContainer container  = builder.Build();

              IReadOnlyList<IPipelineItem> operations =
container.Resolve<IEnumerable<IPipelineItem>>().ToList();

            foreach (IPipelineItem operation in operations)
            {
                Console.WriteLine($"Operation name is {operation.Name}");
            operation.Run();
            }

      }
      }

      public interface IPipelineItem
      {
      string Name { get; }

      void Run();
      }

      public abstract class PipelineItem : IPipelineItem
      {
      public abstract string Name { get; }

      public abstract void Run();
      }

      public sealed class FirstPipelineItem : PipelineItem
      {
      public override string Name => $"{nameof(FirstPipelineItem)}";

      public override void Run()
```

```
        {
                Console.WriteLine($"Hello from {Name}");
        }
        }

        public sealed class SecondPipelineItem : PipelineItem
        {
        public override string Name => $"{nameof(SecondPipelineItem)}";

        public override void Run()
        {
                Console.WriteLine($"Hello from {Name}");
        }
        }

        public sealed class DecoratorOperation : IPipelineItem
        {
        private readonly IPipelineItem _decoratedPipelineItem;

        public DecoratorOperation(IPipelineItem decoratedPipelineItem)
        {
                _decoratedPipelineItem = decoratedPipelineItem;
        }

        public string Name => _decoratedPipelineItem.Name;

        public void Run() => _decoratedPipelineItem.Run();
        }
}
```

Если запустить код, то результат должен быть идентичен предыдущему примеру.

```
Operation name is First pipe item

Hello from FirstPipelineItem

Operation name is Second pipe item

Hello from SecondPipelineItem
```

Композиты и агрегаторы

Зачастую при внедрении зависимостей требуется либо агрегировать зависимости в один новый объект, либо сделать композитную оболочку для множества идентичных объектов. И данный функционал уже встроен в Autofac изначально, что упрощает написание кода в разы.

Возьмем предыдущий пример. Допустим, нужно сделать один единый Pipe, который запускает сразу все IPipelineItem по очереди.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;
using Autofac.Features.Metadata;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();

           builder.RegisterType<FirstPipelineItem>().As<IPipelineItem>();

builder.RegisterType<SecondPipelineItem>().As<IPipelineItem>();

            builder.RegisterComposite<Pipe, IPipelineItem>();

            IContainer container  = builder.Build();

            IPipelineItem pipe = container.Resolve<IPipelineItem>();

            pipe.Run();

      }
      }

      public interface IPipelineItem
      {
        string Name { get; }

      void Run();
      }

      public abstract class PipelineItem : IPipelineItem
      {
      public abstract string Name { get; }

      public abstract void Run();
      }

      public sealed class FirstPipelineItem : PipelineItem
      {
      public override string Name => $"{nameof(FirstPipelineItem)}";

      public override void Run()
      {
            Console.WriteLine($"Hello from {Name}");
      }
      }

    public sealed class SecondPipelineItem : PipelineItem
```

```
        {
        public override string Name => $"{nameof(SecondPipelineItem)}";

        public override void Run()
        {
              Console.WriteLine($"Hello from {Name}");
        }
        }

        public sealed class Pipe : IPipelineItem
        {
        private readonly IEnumerable<IPipelineItem> _items;

        public Pipe(IEnumerable<IPipelineItem> pipelines)
        {
              _items = pipelines;
        }

        public string Name => "Full pipeline";

        public void Run()
        {
              foreach (IPipelineItem item in _items)
              {
              item.Run();
              }
        }
        }
}
```

Ранее зарегистрированные IPipelineItem внедряются в последствии коллекцией IEnumerable<IPipelineItem> в композитный объект.

Но, что делать если у вашего класса слишком много зависимостей? Более пяти или даже шести? Такое бывает при сложном наследовании или когда класс усложнен. Для упрощения можно прибегнуть к агрегатору. Но для этого стоит установить дополнительный nuget пакет Autofac.Extras.AggregateService

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Autofac;
using Autofac.Extras.AggregateService;
using Autofac.Features.Metadata;

namespace Interview.Sample
{
      class Program
      {
      static void  Main(string[] args)
      {
            var builder = new ContainerBuilder();

            builder.RegisterAggregateService<IAggregationService>();
            builder.RegisterType<Logger>().As<ILogger>();

builder.RegisterType<CalculationService>().As<ICalculationService>();

            builder.RegisterType<ComplexObject>();

            IContainer container  = builder.Build();

            ComplexObject complex = container.Resolve<ComplexObject>();

      }
      }

      public interface ILogger
      {

      }

      public sealed class Logger : ILogger
      {

      }

      public interface ICalculationService
      {

      }

      public sealed class CalculationService : ICalculationService
      {

      }

      public interface IAggregationService
      {
      public ILogger Logger { get; }

      public ICalculationService CalculationService { get; }
```

```
        }

        public sealed class ComplexObject
        {
        private readonly IAggregationService _aggregationService;

        public ComplexObject(IAggregationService aggregationService)
        {
                _aggregationService = aggregationService;
        }
        }

}
```

Autofac создаст прокси-класс IAggregationService и передаст ILogger и ICalculationService. Обратите внимание, что создание произойдет автоматически, не надо писать класс-реализацию для IAggregationService.

Практическое задание

1. Используйте предыдущее домашнее задание с эмулятором сканера и по максимуму переведите его на Autofac используя встроенный функционал паттернов и внедрения зависимостей.

Дополнительные материалы

1. Статья «Шаблон проектирования».

Используемые источники

1. Статья «Шаблон проектирования».
2. Статья Design Patterns.
3. Документация по Autofac