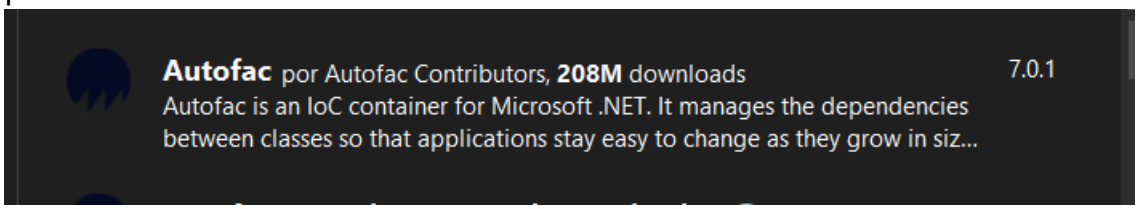
# Autofac

1. **Adicionando o Autofac ao projeto:**
   a. **Adicionando o pacote Autofac:**
   Podemos adicionar o pacote do Autofac com o gerenciador de pacotes NuGet:




   Podemos adicionar o pacote do Autofac através de comando também:

   ➔ Dotnet add package Autofac

   b. **Adicionando container do Autofac no builder:**
   Para usarmos o Autofac devemos adicionar o container do Autofac no nosso builder, isso pode ser feito da seguinte forma:

```
using System;
using Autofac;

namespace DemoApp
{
  public class Program
  {
    private static IContainer Container { get; set; }

    static void Main(string[] args)
    {
      var builder = new ContainerBuilder();
      builder.RegisterType<ConsoleOutput>().As<IOutput>();
      builder.RegisterType<TodayWriter>().As<IDateWriter>();
      Container = builder.Build();

      // The WriteDate method is where we'll make use
      // of our dependency injection. We'll define that
      // in a bit.
      WriteDate();
    }
  }
}
```

   Ou podemos adicionar assim:

```
public static async Task Main(string[] args)
{
  // The service provider factory used here allows for
  // ConfigureContainer to be supported in Startup with
  // a strongly-typed ContainerBuilder.
  var host = Host.CreateDefaultBuilder(args)
    .UseServiceProviderFactory(new AutofacServiceProviderFactory())
    .ConfigureWebHostDefaults(webHostBuilder => {
      webHostBuilder
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
    })
    .Build();

  await host.RunAsync();
}
}
```

Porem dessa forma devemos ter o seguinte package: `Autofac.Extensions.DependencyInjection`

## 2. Conceitos de registro:

Você registra os componentes com o Autofac criando um **ContainerBuilder** e informando ao construtor quais **componentes** expõem quais **serviços**.

Cada componente expõe um ou mais **serviços** conectados usando os **As()** métodos em **ContainerBuilder**.

```
// Create the builder with which components/services are registered.
var builder = new ContainerBuilder();

// Register types that expose interfaces...
builder.RegisterType<ConsoleLogger>().As<ILogger>();

// Register instances of objects you create...
var output = new StringWriter();
builder.RegisterInstance(output).As<TextWriter>();

// Register expressions that execute to create objects...
builder.Register(c => new ConfigReader("mysection")).As<IConfigReader>();

// Build the container to finalize registrations
// and prepare for object resolution.
var container = builder.Build();

// Now you can resolve services using Autofac. For example,
// this line will execute the lambda expression registered
// to the IConfigReader service.
using(var scope = container.BeginLifetimeScope())
{
  var reader = scope.Resolve<IConfigReader>();
}
```

Se você deseja expor um componente como um conjunto de serviços, bem como usar o serviço padrão, use o AsSelfmétodo:

```
builder.RegisterType<CallLogger>()
       .AsSelf()
       .As<ILogger>()
       .As<ICallInterceptor>();
```

É importante destacar que o autofac possui métodos para registro de diferentes tipos de ações como métodos, classes, construtores, interfaces etc. Você pode encontrar mais detalhes sobre eles na documentação: https://autofac.readthedocs.io/en/latest/register/index.html

3. **Tempos de vida de componentes:**

   Diferente do container padrão da Microsoft o Autofac não possui 3 tempos de vida e sim dezenas, são eles:

   **Transient**: Esta é a vida útil padrão e cria uma nova instância do componente sempre que é resolvido.

   **Singleton**: Cria uma única instância do componente que é usada sempre que o componente é resolvido.

   **InstancePerLifetimeScope**: Cria uma única instância do componente por escopo de tempo de vida e reutiliza essa instância sempre que o componente é resolvido dentro desse escopo.

   **InstancePerMatchingLifetimeScope**: Cria uma única instância do componente para cada escopo de tempo de vida que corresponde a um escopo específico fornecido durante a resolução.

   **InstancePerDependency**: Cria uma única instância do componente para cada solicitação de resolução.

   **InstancePerHttpRequest**: Cria uma única instância do componente para cada solicitação HTTP.

   **InstancePerOwned**: Cria uma única instância do componente por proprietário e libera essa instância quando o proprietário é liberado.

**InstancePerLifetimeScopeNamed**: Cria uma única instância do componente para cada escopo de tempo de vida com um nome específico fornecido durante a resolução.

**InstancePerMatchingLifetimeScopeNamed**: Cria uma única instância do componente para cada escopo de tempo de vida com um nome e um escopo específico fornecido durante a resolução.

**InstancePerMatchingLifetimeScopeTagged**: Cria uma única instância do componente para cada escopo de tempo de vida que corresponde a uma tag específica e um escopo específico fornecidos durante a resolução.

**InstancePerMatchingLifetimeScopeTyped**: Cria uma única instância do componente para cada escopo de tempo de vida que corresponde a um tipo específico e um escopo específico fornecidos durante a resolução.

**InstancePerOwned**<T>: Cria uma única instância do componente por proprietário e libera essa instância quando o proprietário é liberado. A diferença em relação ao **InstancePerOwned** é que este tempo de vida especifica o tipo do proprietário.

**InstancePerMatchingLifetimeScope**<T>: Cria uma única instância do componente para cada escopo de tempo de vida que corresponde a um tipo específico fornecido durante a resolução.

**InstancePerMatchingLifetimeScope<TLimit, TActivatorData, TRegistrationStyle>:** Cria uma única instância do componente para cada escopo de tempo de vida que corresponde a um tipo específico, informações de ativação e estilo de registro específicos fornecidos durante a resolução.

O tempo de vida padrão é **Transient**., que significa que uma nova instância do componente é criada cada vez que ele é resolvido pelo contêiner Autofac.

4. **Configuração via JSON/XML:**
   É importante entender que o Autofac pode ser configurado via arquivo e método ConfigurationBuilder():

```
// Add the configuration to the ConfigurationBuilder.
var config = new ConfigurationBuilder();
// config.AddJsonFile comes from Microsoft.Extensions.Configuration.Json
// config.AddXmlFile comes from Microsoft.Extensions.Configuration.Xml
config.AddJsonFile("autofac.json");

// Register the ConfigurationModule with Autofac.
var module = new ConfigurationModule(config.Build());
var builder = new ContainerBuilder();
builder.RegisterModule(module);
```

   O arquivo JSON de configuração poder ser algo parecido com isso:

```
{
  "components": [{
    "type": "Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition",
    "services": [{
      "type": "Autofac.Example.Calculator.Api.IOperation"
    }, {
      "type": "Autofac.Example.Calculator.Api.IAddOperation",
      "key": "add"
    }],
    "autoActivate": true,
    "injectProperties": true,
    "instanceScope": "per-dependency",
    "metadata": [{
      "key": "answer",
      "value": 42,
      "type": "System.Int32, mscorlib"
    }],
    "ownership": "external",
    "parameters": {
      "places": 4
    },
    "properties": {
      "DictionaryProp": {
        "key": "value"
      },
      "ListProp": [1, 2, 3, 4, 5]
    }
  }]
}
```

   É importante entender que o Autofac possui diversas propriedades de configuração e é importante sempre olhar a documentação onde contêm todas essas configurações:
   https://autofac.readthedocs.io/en/latest/configuration/xml.html#

5. **Módulos:**
   O Autofac traz o conceito de módulos que basicamente são uma forma de você organizar a injeção de dependências no container através da localização classes ou arquivos, isso possibilitada separar a logica de injeção de dependências ao longo do código e não apenas no método main().

**a. Módulos via classe:**

Os módulos via classe são os mais comuns e consistem em separar a logica de injeção de dependências em classes:

```
oFacProje                                   AutoFacProje.AutoFac.AutoFacBusiness
1   using Autofac;
2   using AutoFacProje.Business;
3   using System;
4   using System.Collections.Generic;
5   using System.Linq;
6   using System.Threading.Tasks;
7
8   namespace AutoFacProje.AutoFac
9   {
10      public class AutoFacBusiness:Module
11      {
12          protected override void Load(ContainerBuilder builder)
13          {
14              builder.RegisterType<Test>().As<ITest>();
15          }
16      }
17  }
18
```

É importante entender que devemos declarar os módulos no builder:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
    .UseServiceProviderFactory(new AutofacServiceProviderFactory())
    .ConfigureContainer<ContainerBuilder>(builder => {
        builder.RegisterModule(new AutoFacBusiness());
    })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

```
builder.RegisterModule(new CarTransportModule() {
    ObeySpeedLimit = true
});
```

**b. Via configuração:**

Nos podemos declarar os módulos via arquivo de configuração também:

```
{
  "modules": [{
    "type": "MyNamespace.CarTransportModule, MyAssembly",
    "properties": {
      "ObeySpeedLimit": true
    }
  }]
}
```