Dependency Injection of Internals

Part 2

By Andrew Hinkle

Please read the article on [Dependency Injection of Internals in .NET Core 2.1](https://www.danylkoweb.com/Blog/dependency-injection-of-internals-in-net-core-21-OQ) before continuing. That article lays out the concepts of how we got here. I was asked how to expand this concept further by implementing a roll your own IoC container to register dependencies by standard convention (IClassName => ClassName). They also wanted the capability to abstract the manually bindings to a configuration file. As an added benefit I was able to refactor the configuration file binding as a service and moved it down into the internal project as well. Awesome!

My first step involved updating the application from .NET Core 2.1 to .NET Core 3.1. I followed the steps to [Migrate from ASP.NET Core 2.2 to 3.0](https://docs.microsoft.com/en-us/aspnet/core/migration/22-to-30?view=aspnetcore-3.1&tabs=visual-studio). The steps were straight forward though I did have to fix somethings. If interested, you can view the history of changes in my GitHub Tips repository in the [Tips.DependencyInjectionOfInternals](https://github.com/penblade/Tips/tree/master/Tips.DependencyInjectionOfInternals) folder.

Given the number of additional required files for this variation you may no longer want to just copy and paste the couple files to each of your projects. You may isolate the roll your own dependency injection files into a separate project or NuGet package that your internal projects reference. Those files are all in the Configuration folder. However, be aware of the irony this presents. If you go that route you should highly consider implementing a standard IoC package that is maintained with a large supportive community.

If you're still interested in rolling your own IoC container or at least understanding the basics of how they work, then here's my take on it.

# Tips.DependencyInjectionOfInternals.Business

## Configuration\ServiceCollectionForBusiness.cs

The previous version of this class used to register the dependencies manually which is perfectly fine. For small projects that's all you need.

1. Register dependencies by standard convention. IClassName => ClassName.
   1. Reference AssemblyTypes for details.
2. Register the BusinessConfiguration.
   1. Therefore, we don't need to pass it around any more in the Startup and Program.
   2. Handled all within this class using the default [ConfiguraitonBuilder](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.configurationbuilder?view=dotnet-plat-ext-3.1) implementation of [IConfigurationBuilder](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.iconfigurationbuilder?view=dotnet-plat-ext-3.1).
3. Register by configuration file loads the dependencies defined in dependencyConfiguration.json.
   1. Reference the Dependency\* sections for details.

using System;

using System.Collections.Generic;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

public class ServiceCollectionForBusiness : IServiceCollectionForBusiness

{

private IConfiguration \_appSettingsConfiguration;

private readonly IConfigurationBuilder \_configurationBuilder;

public ServiceCollectionForBusiness(IConfiguration appSettingsConfiguration,

IConfigurationBuilder configurationBuilder)

{

\_appSettingsConfiguration = appSettingsConfiguration;

\_configurationBuilder = configurationBuilder;

}

public void RegisterDependencies(IServiceCollection services)

{

RegisterByConvention(services);

var businessConfiguration = RegisterBusinessConfiguration(services);

var dependencyConfig = RegisterDependencyConfiguration(services, businessConfiguration?.IocFiles);

DependencyRegistrar.RegisterDependencyConfiguration(services, dependencyConfig);

}

private static void RegisterByConvention(IServiceCollection services)

{

var assemblyTypes = AssemblyTypes.GetByDefaultConvention();

foreach (var (serviceType, implementationType) in assemblyTypes)

{

services.AddScoped(serviceType, implementationType);

}

}

private BusinessConfiguration RegisterBusinessConfiguration(IServiceCollection services)

{

// Bind the BusinessConfiguration section from the appsettings.json.

const string sectionName = nameof(BusinessConfiguration);

var businessConfiguration = CreateConfigurationByType<BusinessConfiguration>();

\_appSettingsConfiguration.Bind(sectionName, businessConfiguration);

services.AddSingleton(businessConfiguration);

return businessConfiguration;

}

private DependencyConfiguration RegisterDependencyConfiguration(IServiceCollection services,

IEnumerable<string> iocFiles)

{

// Bind the json files. Update the app settings configuration after adding json files.

\_appSettingsConfiguration = AddJsonFiles(\_configurationBuilder, iocFiles);

var dependencyConfig = CreateConfigurationByType<DependencyConfiguration>();

\_appSettingsConfiguration.Bind(dependencyConfig);

services.AddSingleton(dependencyConfig);

return dependencyConfig;

}

private static T CreateConfigurationByType<T>() => (T)Activator.CreateInstance(typeof(T));

private static IConfiguration AddJsonFiles(IConfigurationBuilder configurationBuilder, IEnumerable<string> jsonFiles)

{

if (jsonFiles == null) throw new ArgumentException("No Json files were found.");

foreach (var file in jsonFiles)

{

configurationBuilder.AddJsonFile(file, false, true);

}

return configurationBuilder.Build();

}

}

}

## Configuration\IServiceCollectionForBusiness.cs

I no longer needed to pass in the configuration, just the services.

using Microsoft.Extensions.DependencyInjection;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

public interface IServiceCollectionForBusiness

{

void RegisterDependencies(IServiceCollection services);

}

}

## Configuration\AssemblyTypes.cs

AssemblyTypes registers types by standard convention that most IoC containers perform. Standard convention maps an interface to a single concrete class that has the same name minus the "I" prefix (IClassName => ClassName). This is a roll your own implementation that you can adjust to meet your specific needs. I have not run any metrics to know if it performs better or worse than standard IoC containers, but this is a way to do it.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

internal static class AssemblyTypes

{

public static IEnumerable<(Type serviceType, Type implementationType)> GetByDefaultConvention()

{

var results = new List<(Type, Type)>();

var assembly = Assembly.GetExecutingAssembly();

var assemblyClasses = assembly.GetTypes().Where(x => x.IsClass && !x.IsAbstract).ToList();

var assemblyInterfaces = assembly.GetTypes().Where(x => x.IsInterface).ToList();

// If there are no interfaces or classes, then there is nothing to register.

if (!assemblyInterfaces.Any() || !assemblyClasses.Any()) return results;

// Do this by interface to ignore all of the POCOs.

foreach (var assemblyInterface in assemblyInterfaces)

{

var assemblyClass = assemblyClasses.SingleOrDefault(

x => x.Name == assemblyInterface.Name.TrimStart('I') &&

x.GetInterfaces().FirstOrDefault(y => y.FullName == assemblyInterface.FullName) != null);

// If a class is not found that implements the default conventions, then there is nothing to register.

if (assemblyClass == null) continue;

results.Add((assemblyInterface, assemblyClass));

}

return results;

}

}

}

## Configuration\BusinessConfiguration.cs

Added reference to the IocFiles so we can register dependencies by configuration file.

using System.Collections.Generic;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

public class BusinessConfiguration

{

public string ConnectionString { get; set; }

public string DocumentPath { get; set; }

public List<string> IocFiles { get; set; }

}

}

## Configuration\Dependency.cs

Dependency, DependencyConfiguration, and DependencyRegistrar are used when you want to register dependencies via a configuration file.

The DependencyConfiguration contains a list of Dependency objects. This class is used in the DependencyRegistrar to parse dependencies and register them properly. They represent the data stored in the dependencyConfiguration.json and are loaded via ServiceCollectionForBusiness.RegisterDependencyConfiguration.

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

internal class Dependency

{

public string Namespace { get; set; }

public string ServiceLifetime { get; set; }

public string ServiceType { get; set; }

public string ImplementationType { get; set; }

}

}

## Configuration\DependencyConfiguration.cs

using System.Collections.Generic;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

internal class DependencyConfiguration

{

public string Namespace { get; set; }

public IEnumerable<Dependency> Dependencies { get; set; }

}

}

## dependencyConfiguration.json

{

"Namespace": "Tips.DependencyInjectionOfInternals.Business",

"Dependencies": [

{

"Namespace": "Commands",

"ServiceLifetime": "Scoped",

"ServiceType": "ICommand",

"ImplementationType": "CommandB"

},

{

"Namespace": "Commands",

"ServiceLifetime": "Scoped",

"ServiceType": "ICommand",

"ImplementationType": "CommandA"

},

{

"Namespace": "Commands",

"ServiceLifetime": "Scoped",

"ServiceType": "ICommand",

"ImplementationType": "CommandC"

}

]

}

## Tips.DependencyInjectionOfInternals.Business.csproj

Updated to always output the dependencyConfiguration.json. I ran into problems where the file was not copied naturally.

<ItemGroup>

<None Remove="dependencyConfiguration.json" />

</ItemGroup>

<ItemGroup>

<EmbeddedResource Include="dependencyConfiguration.json">

<CopyToOutputDirectory>Always</CopyToOutputDirectory>

</EmbeddedResource>

</ItemGroup>

## Configuration\DependencyRegistrar.cs

using System;

using Microsoft.Extensions.DependencyInjection;

namespace Tips.DependencyInjectionOfInternals.Business.Configuration

{

internal static class DependencyRegistrar

{

internal static void RegisterDependencyConfiguration(IServiceCollection services, DependencyConfiguration configuration)

{

// Services are returned in the order they were registered in the Startup.

foreach (var dependency in configuration.Dependencies)

{

var (serviceLifetime, serviceType, implementationType) = ParseDependency(configuration.Namespace, dependency);

RegisterDependencyByServiceLifeTime(services, serviceLifetime, serviceType, implementationType);

}

}

private static (ServiceLifetime serviceLifetime, Type serviceType, Type implementationType) ParseDependency(string configurationNamespace, Dependency dependency)

{

var serviceType = ParseType(BuildTypeName(configurationNamespace, dependency.Namespace, dependency.ServiceType));

var implementationType = ParseType(BuildTypeName(configurationNamespace, dependency.Namespace, dependency.ImplementationType));

return (Enum.Parse<ServiceLifetime>(dependency.ServiceLifetime), serviceType, implementationType);

}

private static Type ParseType(string typeName) => Type.GetType(typeName);

private static string BuildTypeName(string namespaceStart, string namespaceEnd, string typeName) => $"{namespaceStart}.{namespaceEnd}.{typeName}";

private static void RegisterDependencyByServiceLifeTime(IServiceCollection services, ServiceLifetime serviceLifetime,

Type serviceType, Type implementationType)

{

switch (serviceLifetime)

{

case ServiceLifetime.Scoped:

services.AddScoped(serviceType, implementationType);

break;

case ServiceLifetime.Transient:

services.AddTransient(serviceType, implementationType);

break;

case ServiceLifetime.Singleton:

services.AddSingleton(serviceType, implementationType);

break;

default:

throw new ArgumentOutOfRangeException(nameof(serviceLifetime), serviceLifetime, "Configuration Error: Dependency service lifetime does not exist.");

}

}

}

}

# Tips.DependencyInjectionOfInternals

## appsettings.json

Added the IocFiles section which identifies the dependency configuration for the business configuration which is registered in ServiceCollectionForBusiness.

{

"BusinessConfiguration": {

"ConnectionString": "Super Secret Database Connection String that should be hidden by managing User Secrets.",

"DocumentPath": "A path to server storage.",

"IocFiles": [

"dependencyConfiguration.json"

]

}

}

## Program.cs

Added the ConfigurationBuilder so I don't have to pass the Configuration all over the place as it is registered at the same time as the services in the ServiceCollectionForBusiness.

namespace Tips.DependencyInjectionOfInternals

{

public class Program

{  
…

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

// Required to create configurations from JSON files.

.ConfigureServices(services => services.AddTransient<IConfigurationBuilder, ConfigurationBuilder>())

.ConfigureServices(services => services.AddTransient<IServiceCollectionForBusiness, ServiceCollectionForBusiness>())

.UseStartup<Startup>();

}

}

## Startup.cs

I no longer needed to pass in the configuration, just the services.

namespace Tips.DependencyInjectionOfInternals

{

public class Startup

{

private readonly IServiceCollectionForBusiness \_serviceCollectionForBusiness;

public Startup(IServiceCollectionForBusiness serviceCollectionForBusiness)

{

\_serviceCollectionForBusiness = serviceCollectionForBusiness;

}

…

private void AddBusinessLibrary(IServiceCollection services)

{

\_serviceCollectionForBusiness.RegisterDependencies(services);

}

## Tips.DependencyInjectionOfInternals.csproj

Updated to always output the appsettings.json. I ran into problems where the file was not copied naturally.

<ItemGroup>

<Content Update="appsettings.json">

<CopyToOutputDirectory>Always</CopyToOutputDirectory>

</Content>

</ItemGroup>

# Conclusion

I demonstrated how you can enhance the original version to register the configuration file in the internal project further isolating the exposure of dependencies. I showed how to register a roll your own IoC container to implement standard conventions. Finally, I enhanced the project to register dependencies based on a configuration file.

*Do you roll your own IoC container? Have you considered or implemented dependency registration from a configuration file? Do you use internals or do you expose all of your classes with public?*