Replace DI Activity

Software education is something we care a lot about at DPL. Doug and I even teach a class on Software Education, it is basically a course to help develop SR Engineers within an organization. We cover a lot of the basic concepts of good software design and construction.

One of those basic concepts is SOLID. It is an acronym created by Robert Martin. Each letter means something different.

S - Single Responsibility

O - Open Closed Principle

L - Liskov Substitution

I - Interface Segregation

D - Dependency Inversion

Here we are going to focus on the concept of Dependency Inversion. Dependency Inversion means that we should have our code depend upon interfaces, not upon the actual services themselves. So, in practice one service should only know about an interface to another service, not the service itself.

To achieve this, we use a basic factory pattern when creating services. Basically, every service will use a factory to gain access to a service, instead of just new’ing up a service. Below is an example of what I am talking about.

using DPLRef.eCommerce.Common.Contracts;

using System;

using System.Collections.Generic;

namespace DPLRef.eCommerce.Common.Shared

{

public abstract class FactoryBase

{

public AmbientContext Context { get; private set; }

protected FactoryBase(AmbientContext context)

{

// context is provided to enable attachment to each instance provided by the factory

Context = context;

}

// contains the dictionary of overrides provided for this factory instance

readonly Dictionary<string, object> \_overrides = new Dictionary<string, object>();

// contains the dictionary of types supported by this factory

readonly Dictionary<string, Type> \_types = new Dictionary<string, Type>();

/// <summary>

/// Provides mock override objects for testing purposes

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="obj"></param>

public void AddOverride<T>(T obj)

{

if (\_overrides.ContainsKey(typeof(T).Name))

\_overrides.Remove((typeof(T).Name));

\_overrides.Add(typeof(T).Name, obj);

}

/// <summary>

/// Configure the types supported by this factory

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="obj"></param>

public void AddType<T>(Type obj)

{

if (\_types.ContainsKey(typeof(T).Name))

\_types.Remove((typeof(T).Name));

\_types.Add(typeof(T).Name, obj);

}

protected T GetInstanceForType<T>() where T : class

{

// Return the override, if one exists for the type T

if (\_overrides.ContainsKey(typeof(T).Name))

{

return \_overrides[typeof(T).Name] as T;

}

// No override, so return an instance of the type from the configured types

if (\_types.ContainsKey(typeof(T).Name))

{

var type = \_types[typeof(T).Name] as Type;

if (type != null)

{

return Activator.CreateInstance(type) as T;

}

}

// Oops, no override OR configuration for this type

throw new ArgumentException($"{typeof(T).Name} is not supported by this factory");

}

}

}

In our class we intentionally have kept as much technology out of it as possible. We are basically using the most basic parts of .NET. We do this because we want to focus on the concepts, and not how to do something with a particular framework.

We did hide away how we implemented DI behind a base factory (FactoryBase). That allows us to change how we implement our factory pattern, without changing the rest of the code. We should be able to change dependency injection technology, without changing any of the other code in our system.

So, to switch our FactoryBase to use .NET Core we should only have to change a couple of methods to use .NET Core’s way of handling dependency injection. And boom we are done.

To do this in .NET Core we have to just use two classes, ServiceProvider and ServiceCollection. To add services to the collection, create a new ServiceDescriptor. Then add that descriptor to the collection.

var service = new ServiceDescriptor(typeof(T), obj, ServiceLifetime.Scoped);

serviceCollection.Replace(service);

To create a service, we must call GetService on the ServiceProvider. To accomplish this, we will create a wrapper method called GetService on our FactoryBase.

private T GetService<T>()

{

if (serviceProvider == null)

{

serviceProvider = serviceCollection.BuildServiceProvider();

}

return serviceProvider.GetService<T>();

}

To create a service from GetInstanceForType call the GetService method we just created.

We could also use unity to do dependency injection. Unity is a library by the Microsoft Patterns and Practices Team, and is a good way to do dependency injection on the .NET Framework. We could have done the same thing using unity. See the example below.

<https://gist.github.com/chadmichel/bd6dd938d9fe2fb927217dcad226ddf5>

Regardless of which technology we use to do dependency injection it is nice to have hidden that choice behind our FactoryBase class. That allows us to change that technology our easily. This is a good example of the single responsibility principle S above. Each module should have only one reason to change. Our FactoryBase has only one reason to change, it will only change if we change our dependency injection.