**Implementing Dependency Injection in ASP.NET Core**

[Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) is the heart of clean architecture for an ASP.NET Core application. It is supported out of the box when you create an ASP.NET Core application with the Microsoft.Extensions.[DependencyInjection](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection?view=dotnet-plat-ext-3.1) module. In the start up file, you can add the mapping between interface and actual implementation. [IServiceCollection](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.iservicecollection) has methods to add services to the container with Lifetime. When the app starts up, services are added to the container. When a class is instantiated, concrete classes are to be injected.

This is really all you need to do to support DI in the ASP.NET core app.

<script src="https://gist.github.com/mydatahack/1d7b957ac462c4345c0fb0b7dbac2a69.js"></script>

The problem with this approach is that we need to keep adding dependencies as we add more classes and interfaces. What if we have hundreds of dependencies and we need to keep track? This solution becomes unmanageable quickly. It is also annoying to update a dependency in another file whenever we create a new class.

**New Solution**

So, here is my solution that I borrowed from [Sitecore’s Habitat example](https://github.com/Sitecore/Habitat/tree/master/src/Foundation/DependencyInjection/code). I have written a blog about [DI in Sitecore](https://www.mydatahack.com/implementing-dependency-injection-in-sitecore/) where I mentions this example. As it is using the out-of-the-box Microsoft Dependency Injection module, it can be used for ASP.NET Core applications.

The idea is very simple. We extend IServiceCollection to have methods to read class attributes with service type and lifetime information by reflection. Then, dependencies get registered when the app starts up. In this way, we can manage dependencies in the same class file by adding attributes instead of keeping in the separate files.

See what I mean below.

We can add attributes with service type and lifetime. See my blog on DI lifetime [here](https://www.mydatahack.com/dependency-injection-lifetime-management-in-sitecore/) for further information.

<script src="https://gist.github.com/mydatahack/eb72d4a325b310b986e24b1723c23a3f.js"></script>

In the startup, we can add a new method with the wildcard keyword for the dll. With the extended method, it will look for the service attribute in all the classes including the word passed as an argument. Look how manageable the Startup.cs file becomes!

<script src="https://gist.github.com/mydatahack/2e12763e01f2037456fc262396a48264.js"></script>

**Implementing IServiceCollection Extensions**

Let’s dive deep into the actual implementation. As I mentioned above, I adapted this solution from [Sitecore’s Habitat example](https://github.com/Sitecore/Habitat/tree/master/src/Foundation/DependencyInjection/code). There are a bit of changes. If you copy & paste this solution, it will work for ASP.NET Core apps. One of the biggest difference is that we do not need to register controllers explicitly as we can use AddMvc() to register controllers.

If your solution has only one project, you can create a folder called Extensions and add them in there. If you are using an Onion architecture like [this example](https://github.com/dotnet-architecture/eShopOnWeb), I would put it in the Application Core as it needs to be referenced by other projects. It is however an anti-pattern as this extension is not really a business logic.

So, once you have the Extension folder, you need to create two files. One is for the attribute model and the other is the actual extensions.

**ServiceAttribute.cs**

<script src="https://gist.github.com/mydatahack/85ca290c45a1cb8ee5ee21920000b54f.js"></script>

**SearviceCollectionExtenstions.cs**

<script src="https://gist.github.com/mydatahack/17f37e53a38d2869102398ea82e9ba90.js"></script>

Once you add these methods, you can start using attributes to register dependencies in your ASP.NET Core application.

Sweet!

Wait, there is more… What about writing unit tests on this extension method?

**Unit Tests**

First, we create mock classes and interfaces in the test project.

**MockClassForExtensionsTests.cs**

<script src="https://gist.github.com/mydatahack/c2e2b75c4680f8ccc5cf29b28c7af5bd.js"></script>

Then, we can instantiate ServiceCollection and call the newly created extension method, AddClassesWithServiceAttribute and target dll in the unit test project. This will read attributes from the mock classes created above from the dll.

**ServiceCollectionExtensionsTest.cs**

<script src="https://gist.github.com/mydatahack/f675974901f950c140181fec3350cb1a.js"></script>

This is it for now. It is your turn to try this out. Let us know how you go!

**GitGist**

**ServiceAttribute.cs**

<script src="https://gist.github.com/mydatahack/85ca290c45a1cb8ee5ee21920000b54f.js"></script>

**SearviceCollectionExtenstions.cs**

<script src="https://gist.github.com/mydatahack/17f37e53a38d2869102398ea82e9ba90.js"></script>

**Startup - with ServiceCollectionExtension.cs**

<script src="https://gist.github.com/mydatahack/2e12763e01f2037456fc262396a48264.js"></script>

**Startup – out of the box.cs**

<script src="https://gist.github.com/mydatahack/1d7b957ac462c4345c0fb0b7dbac2a69.js"></script>

**Using Attribute example**

<script src="https://gist.github.com/mydatahack/eb72d4a325b310b986e24b1723c23a3f.js"></script>