Dependency Injection of Internals in .NET Core 2.1

By Andrew Hinkle

When working with Legacy applications you'll find class libraries littered with public classes implemented by other class libraries that were never meant for consumption. To mitigate the object leakage developers change public classes to [internal](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal) and expose their usage via a few limited public classes. That's great, but how do you inject internal dependencies into the class library when the dependency injection occurs from the entry project such as an API? Let's start with the basics and work our way to the solution.

Taking advantage of "[poor man's dependency injection](https://softwareengineering.stackexchange.com/questions/364090/is-poor-mans-dependency-injection-a-good-way-to-introduce-testability-to-a-lega)" (PMDI) all of your internal classes will have two constructors. The public constructor expects an implementation of an interface and an internal constructor that creates a new instance of the implementation. This tightly couples the implementation details with the class. The best part is that you can [add unit tests that inject mock versions](https://stackoverflow.com/questions/20423714/mocking-using-moq-in-c-sharp) ([TestDoubles](https://martinfowler.com/bliki/TestDouble.html)) of the interfaces such as repositories, configuration settings, etc. This allows you to test just the logic in the class and not what is injected.

Of course you can take it to the next stage and remove the second constructor causing the tight coupling. To accomplish this feat you now move the creation of the implementation in the calling class. At this point you've just moved the cheese up a layer that now maintains that dependency. In order to take full advantage of this we introduce [Inversion of Control frameworks](https://stackoverflow.com/questions/21288/which-net-dependency-injection-frameworks-are-worth-looking-into).

Here's the rub, once you get out of the class library, you can't use the internal classes you want to use for the implementation. Sure, you could change them to public classes, but now your library is naked for all to see. You could update the class library properties to add an [InternalsVisibleTo](https://stackoverflow.com/questions/42235401/unit-testing-internal-methods-in-vs2017-net-standard-library) to the calling assembly, but that defeats the whole purpose given the tight coupling to the assemblies. Perhaps one of the IOC frameworks can handle registering internals, but I haven't found it.

While working with .NET Core 2.1 Dependency Injection I ran into the same problem yet again. After reviewing [many](https://medium.com/@mattmazzola/asp-net-core-injecting-custom-data-classes-into-startup-classs-constructor-and-configure-method-7cc146f00afb) [articles](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1) and [stackoverflow](https://stackoverflow.com/questions/49703773/implement-ado-connection-in-asp-net-core) posts, I finally came up with a solution I like. The crux of it is to create a public class called ServiceCollectionForBusiness (For {project name}) and interface in the class library that will register the dependencies. Here's the tricky part, you need to inject that class into the Program.cs, so it can be injected into the Startup.cs, and so it can then register the class library internal dependencies. Sounds a bit like injection inception?

In the interest of code overload, I've created a sample WebApi .NET Core 2.1 that demonstrates the concept titled [DependencyInjectionOfInternals](https://github.com/penblade/Tips/tree/master/Tips.DependencyInjectionOfInternals). This simple app has a class library that processes internal CommandA, CommandB, and CommandC that implement ICommand. Note that these commands have external dependency of BusinessConfiguration that we'll need to inject later.

# CommandA

internal class CommandA : ICommand

{

private readonly BusinessConfiguration \_businessConfiguration;

public CommandA(BusinessConfiguration businessConfiguration)

{

\_businessConfiguration = businessConfiguration;

}

public string Process(ProcessRequest request)

{

// Do something

var connectionString = \_businessConfiguration.ConnectionString;

// Return a message to demonstrate when the command was processed.

return $@"CommandA was processed. ConnectionString: {connectionString}";

}

}

# CommandB

internal class CommandB : ICommand

{

public string Process(ProcessRequest request)

{

// Do something

// Return a message to demonstrate when the command was processed.

return "CommandB was processed";

}

}

# CommandC

internal class CommandC : ICommand

{

private readonly BusinessConfiguration \_businessConfiguration;

public CommandC(BusinessConfiguration businessConfiguration)

{

\_businessConfiguration = businessConfiguration;

}

public string Process(ProcessRequest request)

{

// Do something

var documentPath = \_businessConfiguration.DocumentPath;

// Return a message to demonstrate when the command was processed.

return $@"CommandC was processed. DocumentPath: {documentPath}";

}

}

# ICommand

internal interface ICommand

{

string Process(ProcessRequest request);

}

The Commands are created via an internal CommandFactory (an abstract factory) that uses the .NET Core ServiceProvider to return the list of all Commands or a specific ICommand given the CommandType.

# CommandFactory

internal class CommandFactory : ICommandFactory

{

private readonly IServiceProvider \_serviceProvider;

public CommandFactory(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public IEnumerable<ICommand> Create(CommandType commandType)

{

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

var commands = \_serviceProvider.GetServices<ICommand>();

commands = commandType == CommandType.All

? commands

: commands.Where(x => x.GetType().Name.Equals(commandType.ToString()));

return commands;

}

}

# ICommandFactory

internal interface ICommandFactory

{

IEnumerable<ICommand> Create(CommandType commandType);

}

These Commands are the core of this feature and nothing outside of the class library needs to know these internal implementations. The functionality is exposed via the BusinessService class. Here, the internal CommandFactory is injected. Typically, this class would have to be public following the PMDI technique as mentioned earlier. Without the capability of injecting internals the class library would have to be all PMDI violating the principles of a complete DI solution where every class has a single constructor.

# BusinessService

internal class BusinessService : IBusinessService

{

private readonly ICommandFactory \_commandFactory;

public BusinessService(ICommandFactory commandFactory)

{

\_commandFactory = commandFactory;

}

public ProcessResponse Process(ProcessRequest request)

{

var response = new ProcessResponse();

foreach (var command in \_commandFactory.Create(request.CommandType))

{

response.Messages.Add(command.Process(request));

}

return response;

}

}

In order for the API project to use this class library the IBusinessService interface is made public. The only way to use the BusinessService class is by using DI.

# IBusinessService

public interface IBusinessService

{

ProcessResponse Process(ProcessRequest request);

}

I favor leaving POCOs as POCOs without interfaces, so the following public POCOs are exposed for use with IBusinessService. If there were POCOs that were only used within the class library, they would be internal.

# CommandType

public enum CommandType

{

None = 0,

All = 1,

CommandA = 2,

CommandB = 3,

CommandC = 4,

}

# ProcessRequest

public class ProcessRequest

{

public CommandType CommandType { get; set; }

}

# ProcessResponse

public class ProcessResponse

{

public ProcessResponse()

{

Messages = new List<string>();

}

// Leave the setter public for deserialization.

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

}

As mentioned earlier, the class library has an external dependency on the appsettings.json represented under the BusinessConfiguration section by a ConnectionString and DocumentPath.

# BusinessConfiguration

public class BusinessConfiguration

{

public string ConnectionString { get; set; }

public string DocumentPath { get; set; }

}

The goal with these next classes is to separate the concerns of dependency injection. We want the ability to add a class library to the solution and register a single class from that library that knows how to register all of its dependencies. This leaves the Startup clean with a few registrations making it easier to read and maintain.

# ServiceCollectionForBusiness

public class ServiceCollectionForBusiness : IServiceCollectionForBusiness

{

public void RegisterDependencies(IConfiguration configuration,

IServiceCollection services)

{

// Bind the configuration to

var config = new BusinessConfiguration();

configuration.Bind(nameof(BusinessConfiguration), config);

services.AddSingleton(config);

// Setup relationship between public interfaces and internal classes

services.AddScoped<IBusinessService, BusinessService>();

// Setup relationship between internal interfaces and internal classes

services.AddScoped<ICommandFactory, CommandFactory>();

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

services.AddScoped<ICommand, CommandB>();

services.AddScoped<ICommand, CommandA>();

services.AddScoped<ICommand, CommandC>();

}

}

# IServiceCollectionForBusiness

public interface IServiceCollectionForBusiness

{

void RegisterDependencies(IConfiguration configuration,

IServiceCollection services);

}

Now we switch over to the API project and update the Program.cs to configure these services before the Startup.cs.

# Program

public class Program

{

public static void Main(string[] args)

{

CreateWebHostBuilder(args).Build().Run();

}

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

WebHost.CreateDefaultBuilder(args)

// Add custom service collection registration.

// In order to inject classes into the

// Startup constuctor, the services must

// be registered before we UseStartup.

// ConfigureServices must be called before

// UseStartup method. This is how .NET Core

// works under the covers as noted in:

// [Asp.Net Core: Injecting custom](https://medium.com/@mattmazzola/asp-net-core-injecting-custom-data-classes-into-startup-classs-constructor-and-configure-method-7cc146f00afb)

// data/classes into startup classes

// constructor and configure method

// by Matt Mazzola.

.ConfigureServices(services =>

services.AddTransient<

IServiceCollectionForBusiness,

ServiceCollectionForBusiness>())

.UseStartup<Startup>();

}

This gives us the awesome ability to inject the IServiceCollectionForBusiness into the Startup.

# Startup

public class Startup

{

private readonly IServiceCollectionForBusiness \_serviceCollectionForBusiness;

public Startup(IConfiguration configuration,

IServiceCollectionForBusiness serviceCollectionForBusiness)

{

\_serviceCollectionForBusiness = serviceCollectionForBusiness;

Configuration = configuration;

}

public IConfiguration Configuration { get; }

// This method gets called by the runtime.

// Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

AddMvc(services);

AddBusinessLibrary(services);

}

private static void AddMvc(IServiceCollection services)

{

…

}

private void AddBusinessLibrary(IServiceCollection services)

{

\_serviceCollectionForBusiness.RegisterDependencies(Configuration, services);

}

// This method gets called by the runtime.

// Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

…

}

}

Here's the controller that kicks off all of the DI gloriousness.

# CommandController

[ApiController]

public class CommandController : ControllerBase

{

private readonly IBusinessService \_businessService;

public CommandController(IBusinessService businessService)

{

\_businessService = businessService;

}

[Route("v1/api/commands/{commandType=1}")]

[HttpGet]

public ActionResult Get(int commandType)

{

var request = new ProcessRequest { CommandType = (CommandType)commandType };

var response = \_businessService.Process(request);

return Ok(response);

}

}

The appsettings.json is pretty straightforward. It uses dummy data here. Legit secrets should be hidden away from source control by [Managing User Secrets in .NET Core 2.0 Apps](https://blogs.msdn.microsoft.com/mihansen/2017/09/10/managing-secrets-in-net-core-2-0-apps/).

# appsettings.json

{

"BusinessConfiguration": {

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

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

}

}

Sprinkle in some unit tests for the internals class library.

# ServiceCollectionForBusinessTest

[TestClass]

public class ServiceCollectionForBusinessTest

{

private readonly ServiceProvider \_serviceProvider;

// Only use [ClassInitialize] when the class properties

// should only be initalized once for all tests.

// If the test methods change the properties,

// use [TestInitialize] or add a public constructor.

// If you're not sure, just use the unit test's

// public constructor.

public ServiceCollectionForBusinessTest()

{

// The mock configuration must setup all of the

// expected properties or an exception is thrown.

// System.ArgumentNullException: Value cannot be null.

// Value cannot be null.\r\nParameter name: configuration

var mockConfigurationSection = new Mock<IConfigurationSection>();

mockConfigurationSection.SetupGet(x => x[It.IsAny<string>()]).Returns("expected configuration value");

var mockConfiguration = new Mock<IConfiguration>();

mockConfiguration.Setup(x => x.GetSection(It.IsAny<string>())).Returns(mockConfigurationSection.Object);

var serviceCollection = new ServiceCollection();

var serviceCollectionForBusiness = new ServiceCollectionForBusiness();

serviceCollectionForBusiness.RegisterDependencies(mockConfiguration.Object, serviceCollection);

\_serviceProvider = serviceCollection.BuildServiceProvider();

}

[TestMethod]

public void VerifyRegisterDependenciesForBusiness()

{

Assert.IsInstanceOfType(\_serviceProvider.GetService<IBusinessService>(), typeof(BusinessService));

Assert.IsInstanceOfType(\_serviceProvider.GetService<ICommandFactory>(), typeof(CommandFactory));

AssertCommandsWereRegistered();

}

private void AssertCommandsWereRegistered()

{

var expectedCommands = new List<Type> { typeof(CommandB), typeof(CommandA), typeof(CommandC) };

var actualCommands = \_serviceProvider.GetServices<ICommand>().ToList();

Assert.IsNotNull(actualCommands);

Assert.AreEqual(expectedCommands.Count, actualCommands.Count);

for (var i = 0; i < actualCommands.Count; i++)

{

Assert.IsInstanceOfType(actualCommands[i], expectedCommands[i]);

}

}

}

And a dash of unit tests for the API.

# StartupTest

[TestClass]

public class StartupTest

{

// Requires AspNetCore dependency for the

// ServiceCollection to call the static

// extension method AddMvc().

[TestMethod]

public void VerifyRegisterDependenciesForBusinessWasRegistered()

{

// Requires AspNetCore dependency for the

// ServiceCollection to call the static

// extension method AddMvc().

var serviceCollection = new ServiceCollection();

var mockConfiguration = new Mock<IConfiguration>();

var mockServiceCollectionForBusiness = new Mock<IServiceCollectionForBusiness>();

mockServiceCollectionForBusiness.Setup(x => x.RegisterDependencies(mockConfiguration.Object, serviceCollection));

var startup = new Startup(mockConfiguration.Object, mockServiceCollectionForBusiness.Object);

startup.ConfigureServices(serviceCollection);

// Verify that the static method was called once.

mockServiceCollectionForBusiness.Verify(x => x.RegisterDependencies(mockConfiguration.Object, serviceCollection), Times.Once);

}

}

Of course you still need to run an integration test to make sure everything was hooked up correctly. Here's the result for each CommandType.

# None: /v1/api/commands/0

{

"messages": []

}

# All: /v1/api/commands/1

{

"messages": [

"CommandB was processed",

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

"CommandC was processed. DocumentPath: A path to server storage."

]

}

# CommandA: /v1/api/commands/2

{

"messages": [

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

]

}

# CommandB: /v1/api/commands/3

{

"messages": [

"CommandB was processed"

]

}

# CommandC: /v1/api/commands/4

{

"messages": [

"CommandC was processed. DocumentPath: A path to server storage."

]

}

The entire application is available on GitHub under [DependencyInjectionOfInternals](https://github.com/penblade/Tips/tree/master/Tips.DependencyInjectionOfInternals).

# Conclusion

I've shown how we can separate the concerns of dependency injection by implementing that logic in the class library. While this makes it possible to use .NET Core to dependency inject your internal classes, if you prefer to keep everything public, this technique still works. Do you create internal classes or do you keep everything public? Do you prefer to only implement internal classes when the project can be consumed externally via NuGet package or by a third party? Do you like the concept of separating the concerns of dependency inject down into the class libraries? Let's discuss!