Assembly Encapsulation with Dependency Injection – Unit Testing

By Andrew Hinkle

This article assumes you have already read [Assembly Encapsulation with Dependency Injection](https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/docs/Tips.AssemblyEncapsulation/Tips.AssemblyEncapsulationWithDependencyInjection.docx). Please read it for the following discussion to make sense.

Goals of this article:

1. Provide an overview to refresh your memory on the main points of [Assembly Encapsulation with Dependency Injection](https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/docs/Tips.AssemblyEncapsulation/Tips.AssemblyEncapsulationWithDependencyInjection.docx).
2. Describe the unit test pattern used to test each project registered their dependency correctly and provide an example with the Middleware test project.
3. Startup Project Registration Tests section reviews issues with testing the registration of ALL dependencies from the startup project given many of them could be internal and not accessible.
4. Review possible adjustments that will test it if you want and an argument for not testing given indirect test coverage from manual and automated integration tests.

There's a lot here, so let's review.

# Assembly Encapsulation with Dependency Injection overview

1. Assembly Encapsulation is accomplished by having the startup project's Startup ConfigureServices method call an extension method called ServiceCollectionExtensions.RegisterDependencies.
2. ServiceCollectionExtensions.RegisterDependencies calls each dependent project's DependencyRegistrar.Register static method which registers its interfaces and abstract classes to concrete classes.
3. Given the registration happens inside the project, internal interfaces, internal abstract classes, and internal concrete classes can be registered.

# Unit Tests

The unit tests in general follow the pattern of:

* Public class {Class under test}Test
  + Public void {Method under test}Test()

var serviceCollection = new ServiceCollection();

DependencyRegistrar.Register(serviceCollection, configurationRootFromJson);

var serviceProvider = serviceCollection.BuildServiceProvider();

DependencyRegistrarSupport.AssertServiceIsInstanceOfType<ProblemDetailsConfiguration, ProblemDetailsConfiguration>(serviceProvider);

DependencyRegistrarSupport.AssertServiceIsInstanceOfType<IProblemDetailsFactory, ProblemDetailsFactory>(serviceProvider);

DependencyRegistrarSupport.AssertServiceIsInstanceOfType gets the service by type and then asserts its an instance of that type. This was refactored to a support class for reusability.

var service = serviceProvider.GetService<TServiceType>();

Assert.IsInstanceOfType(service, typeof(TImplementationsType));

# Tips.Middleware.Tests.Configuration.DependencyRegistrarTest

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware.Tests/Configuration/DependencyRegistrarTest.cs>

using System;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Support.Tests;

using Tips.Middleware.Configuration;

using Tips.Middleware.ErrorHandling;

namespace Middleware.Tests.Configuration

{

[TestClass]

public class DependencyRegistrarTest

{

[TestMethod]

public void RegisterTest()

{

var configurationRootFromJson = new ConfigurationBuilder().AddJsonFile(@"Configuration\appsettings.test.json").Build();

var serviceCollection = new ServiceCollection();

DependencyRegistrar.Register(serviceCollection, configurationRootFromJson);

var serviceProvider = serviceCollection.BuildServiceProvider();

DependencyRegistrarSupport.AssertServiceIsInstanceOfType<ProblemDetailsConfiguration, ProblemDetailsConfiguration>(serviceProvider);

DependencyRegistrarSupport.AssertServiceIsInstanceOfType<IProblemDetailsFactory, ProblemDetailsFactory>(serviceProvider);

AssertConfiguration(serviceProvider);

}

private static void AssertConfiguration(IServiceProvider serviceProvider)

{

var expectedConfiguration = CreateProblemDetailsConfiguration();

var actualConfiguration = serviceProvider.GetService<ProblemDetailsConfiguration>();

Assert.AreEqual(expectedConfiguration?.UrnName, actualConfiguration?.UrnName);

}

private static ProblemDetailsConfiguration CreateProblemDetailsConfiguration() =>

new()

{

UrnName = "TestUrnName"

};

}

}

# Tips.Middleware.Tests.Configuration.appsettings.test.json

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware.Tests/Configuration/appsettings.test.json>

{

"ProblemDetailsConfiguration": {

"UrnName": "TestUrnName"

}

}

# Tips.Support.Tests

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Support.Tests/DependencyRegistrarSupport.cs>

using System;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Logging;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Support.Tests

{

public static class DependencyRegistrarSupport

{

public static void AddScopedLogger(IServiceCollection serviceCollection) => serviceCollection.AddScoped(typeof(ILogger<>), typeof(FakeLogger<>));

public static void AssertServiceIsInstanceOfType<TServiceType, TImplementationsType>(IServiceProvider serviceProvider)

{

var service = serviceProvider.GetService<TServiceType>();

Assert.IsInstanceOfType(service, typeof(TImplementationsType));

}

public static void AssertServiceIsNotNull<TServiceType>(IServiceProvider serviceProvider)

{

var service = serviceProvider.GetService<TServiceType>();

Assert.IsNotNull(service);

}

}

}

# Startup Project Registration Tests

In each test project I was able to assert that everything was registered properly from within that project. The Startup project is the exception to the rule.

## Tips.Api.Tests.Configuration.ServiceCollectionExtensionsTest

This test class resides in the startup project's tests project (Api.Tests). The goal here would be to verify the dependencies were registered or in the least ensure that each dependent project's RegisterDependencies was called.

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Api.Tests/Configuration/ServiceCollectionExtensionsTest.cs#L19-L32>

namespace Api.Tests.Configuration

{

[TestClass]

public class ServiceCollectionExtensionsTest

{

[TestMethod]

public void RegisterDependencies()

{

var configurationRootFromJson = new ConfigurationBuilder().AddJsonFile(@"Configuration\appSettings.test.json").Build();

var serviceCollection = new ServiceCollection();

DependencyRegistrarSupport.AddScopedLogger(serviceCollection);

serviceCollection.RegisterDependencies(configurationRootFromJson);

var serviceProvider = serviceCollection.BuildServiceProvider();

…

}

Public classes, abstract classes, and interface combinations are easy to assert with actual implementations or mocks. Throw internal in the mix and we just don't have the access, as intended. Let's go through the issues and potential solutions.

## Add [assembly: InternalsVisibleTo("Api.Tests")] to each test project?

Of course, the internals from other assemblies are not accessible here. Adding InternalsVisibleTo("Api.Tests") to the other unit test projects (ex. Tips.Middleware.Tests) also works, and now you can duplicate the same asserts in the other test projects here to verify everything is registered.

using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("Api.Tests")]

[assembly: InternalsVisibleTo("DynamicProxyGenAssembly2")]

Note: [assembly: InternalsVisibleTo("DynamicProxyGenAssembly2")] is for Moq.

While this works, it adds an unnecessary dependency with maintenance consequences.

1. If I rename the startup project test, I'd have to update the magic strings "Api.Tests" in multiple projects.
2. Tips.Middleware is a project meant to be reused in every API. Each time I create a new API that uses this project, I'd have to add yet another startup project reference here.

Nope, not interested in this upkeep and dependency.

## Add each test project as a project reference to the Startup project's test project?

If we did this, we could call each test project's unit test to assert the dependencies were registered since they're already doing it.

This works if the test project is in your solution. If the project is a NuGet package, you would not unless the NuGet package author also included the test project.

## Change static methods to concrete classes?

Mocking RegisterDependencies is out of the equation because these are static methods.

We could convert each Registrar to a concrete class and manually inject them into the Program.cs. This would give us an implementation to confirm exists and assume we're asserting the dependencies in each project which we are.

This would work, but we're adding complication to Program.cs.

## Unit test with integration?

Test the implementation in your unit test and assert the results.

This works. However, if the integration is to a database, you're verifying the commands and reads were done correctly against the real thing. This is fine if you understand the integration and properly setup/reset each test.

## Should I unit test this in the Startup project's test project?

There's a good question. The dependent projects are already testing the dependencies were registered. Any manual or automated integration tests worth their salt should be testing enough of the implementation to confirm that the dependencies were registered. And yes, you should have manual and automated tests to confirm your integrations.

## So, what should I do?

Keep it simple. Not everything needs tested directly. This functionality should be covered by manual and automated integration tests. Only worry about this level of testing if you run into bugs with this logic.

# Unit Test what is critical

As a rule of thumb, unit test critical logic and typically anything discovered as a bug to verify the issue is fixed. Create integration unit tests to verify everything was hooked up. If the infrastructure is critical then unit test it. If the rules are critical, then unit test the rules.

You should have integration tests that test the actual deployed application as well. Remember to perform integration tests, manual and automated to ensure your app works as expected and is hooked up properly.

There's a lot to be said on unit tests and there are plenty of perspectives on what should and should not be tested. I encourage you to do your own research and form your own conclusions.

# Conclusion

I've shown you how to unit test your dependency registration for each project. The exception to the rule is the Startup project since it does not have access to the other project's internal registrations. I provided some guidance on adjustments that would allow these registrations to be tested with pros and cons. I recommended that the tests at this level may not be necessary if you have manual or automated integration tests that already cover it. I ended on the advice to unit test what is critical and to do your own research and form your own conclusions.

*Do you think it is worth unit testing any dependency registrations? Does it make sense for the dependent projects, but not the Startup project?*