Assembly Encapsulation with Dependency Injection

By Andrew Hinkle

[C# is an Object-Oriented programming language](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/) based on the core principles of [Abstraction, Inheritance](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance), [Encapsulation](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/#encapsulation), and [Polymorphism](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism). Encapsulation is enforced with [access modifiers](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels) (public, protected, internal, protected internal, private, and private protected).

Let's focus on public and internal classes. Public classes are accessible from any assembly, so they can typically be derived from and initialized on demand with the new keyword. Internal classes are only accessible from within its assembly allowing you to use the class as you desire without fear that the class will be derived from or initialized outside of the assembly. This is great for locking down your assembly so consumers of your assembly may only access or implement the classes you want exposed.

[Dependency injection (DI)](https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection) is a technique for achieving [Inversion of Control (IoC)](https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion) between classes and their dependencies. Create public abstract classes and interfaces with public methods that expose functionality. Following the basic [dependency injection tutorial](https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-usage) to register the public abstract classes and interfaces with public concrete implementations. With constructor injection any constructor referencing the abstract class or interface will instead use the concrete implementation.

For legacy apps that don't support an IoC container, I've used [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) with the supporting constructor that initializes the dependency. It doesn't truly remove the dependency, but can assist in the refactoring as I strangle the old code out to new assemblies where I could take advantage of a more [Pure DI](https://softwareengineering.stackexchange.com/questions/364090/is-poor-mans-dependency-injection-a-good-way-to-introduce-testability-to-a-lega) solution until an IoC can be implemented.

However, using any of the common IoC containers to service your Dependency Injection needs typically breaks Assembly Encapsulation as most cases involve making everything public to work. What can we do to take advantage of DI so we're not using the "new" keyword all over our assembly just to enforce Assembly Encapsulation? Can we have the best of both worlds, Assembly Encapsulation WITH Dependency Injection? The answer is yes. I'll show you how and provide links to my previous articles that will provide other flavors of this concept.

For this article, we're using a very heavily modified [Todo API](https://github.com/penblade/Tips/tree/master/Tips.ApiMessage/src) originally based on the standard [Tutorial: Create a web API with ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-5.0&tabs=visual-studio) Todo API. After all, real world apps are typically way different than the basic tutorials. Don't worry about all the differences or even how the API Key security is implemented except as noted. We'll be reviewing those differences in subsequent articles.

# Let's start with the scenarios we want to support with examples.

1. Configuration objects specific to the assembly should only be accessible from within the assembly.
2. Abstract class/Interface are exposed publicly, and concrete class is ONLY used in the assembly.
   1. Public interface (IProblemDetailsFactory)
   2. Internal class (ProblemDetailsFactory)
   3. The Middleware project exposed the interface so the API project may call the BadRequest and InternalServerError methods without knowing or caring about how it was implemented.
3. Abstract class/Interface and concrete class are ONLY used in the assembly
   1. Internal interface (ICreateTodoItemRepository)
   2. Internal class (CreateTodoItemRepository)
   3. This is a repository implementation that performs CRUD. The TodoItems project has sole control over how the repository is implemented and nothing outside the assembly should even know it exists. If it wasn't for the fact it even needs registered in the IoC the startup project wouldn't have known either.

# How are we going to accomplish this lofty goal?

We're going to build upon my earlier articles. I explain the concepts well enough in this article, but if you're interested in the journey and alternative implementations, then please check out these articles.

1. Assembly Encapsulation: [Dependency Injection of Internals in .NET Core 2.1](https://www.danylkoweb.com/Blog/dependency-injection-of-internals-in-net-core-21-OQ)
2. Assembly Encapsulation: [Dependency Injection of Internals, Part 2](https://www.danylkoweb.com/Blog/dependency-injection-of-internals-part-2-HB)

The startup project is going to ask each project to register their own dependencies.

1. The startup project does not need to know about any of the project dependencies.
2. Each project has a DependencyRegistrar.Register static method that performs the registration.
3. Pass into the static Register method the service collection and optionally the configuration.
   1. Alternatives and other enhancements to using static registrar classes were discussed in the previous articles.
      1. Example: Roll your own IoC.
      2. Example: Registration by json configuration file
      3. Example: Registration by standardization where the interface name is the same as the concrete class name with an "I" in front (IMyClass to MyClass).
4. Use the access modifiers as intended.
   1. If a class should be internal, then it should be internal!

With goals defined, let's start with a simpler example showing off the concepts with the Middleware project in the sample app and then build upon those concepts into ToDo Items repository and rules example. The Middleware project referenced here is a project I created to support common Middleware functionality.

Source Code here: <https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/>

# Tips.Api.Startup

Inject the Configuration into the Startup class via constructor injection. Call the IServiceCollection extension method RegisterDependencies.

<https://github.com/penblade/Tips/blob/801a32aa4444d73e7df188e35cf5588c8818fca2/Tips.ApiMessage/src/Api/Startup.cs#L16-L26>

namespace Tips.Api

{

public class Startup

{

**private readonly IConfiguration \_configuration;**

**public Startup(IConfiguration configuration)**

**{**

**\_configuration = configuration;**

**}**

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

**services.RegisterDependencies(\_configuration);**

services.AddControllers();

services.AddSwaggerWithApiKeySecurity(\_configuration, $"{Assembly.GetExecutingAssembly().GetName().Name}");

}  
…

# Tips.Api.Configuration.ServiceCollectionExtensions

The crux of this technique is for each assembly to create a static method that registers its own dependencies. RegisterDependencies calls each assembly's DependencyRegistrar.Register method passing in the server collection and optionally the configuration.

With .NET 5 (Core) this takes advantage of the built in IoC. The easiest and most straight forward mechanism is to register the dependencies yourself. This technique can be modified to take advantage of other patterns and IoC containers. There are references at the start and end of the article on this.

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Api/Configuration/ServiceCollectionExtensions.cs>

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

namespace Tips.Api.Configuration

{

public static class ServiceCollectionExtensions

{

public static void RegisterDependencies(this IServiceCollection services, IConfiguration configuration)

{

Pipeline.Configuration.DependencyRegistrar.Register(services);

Middleware.Configuration.DependencyRegistrar.Register(services, configuration);

Security.Configuration.DependencyRegistrar.Register(services, configuration);

Rules.Configuration.DependencyRegistrar.Register(services);

TodoItems.Configuration.DependencyRegistrar.Register(services);

}

}

}

# Add NuGet package for dependency injection and configuration binder:

As a reference I added both to this project: Tips.Middleware.Middleware.csproj

Add the following NuGet Package reference through your method of choice: the Package Manager, Package Manager Console, or manually update the .csproj.

* To register dependencies add: Microsoft.Extensions.DependencyInjection.Abstractions
* To register configuration add: Microsoft.Extensions.Configuration.Binder

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware/Middleware.csproj>

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>net5.0</TargetFramework>

<AssemblyName>Tips.Middleware</AssemblyName>

<RootNamespace>Tips.Middleware</RootNamespace>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.Mvc.Core" Version="2.2.5" />

**<PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="5.0.0" />**

**<PackageReference Include="Microsoft.Extensions.DependencyInjection.Abstractions" Version="5.0.0" />**

<PackageReference Include="Microsoft.Extensions.Logging.Abstractions" Version="5.0.0" />

</ItemGroup>

<ItemGroup>

<ProjectReference Include="..\Pipeline\Pipeline.csproj" />

<ProjectReference Include="..\Security\Security.csproj" />

</ItemGroup>

</Project>

# Tips.Middleware.Configuration.DependencyRegistrar

Since we are now in the Middleware project, we have access to the internal class ProblemDetailsConfiguration. We bind it to the configuration and register it as a singleton. ProblemDetailsConfiguration is now registered in the IoC and is only accessible while in this assembly.

IProblemDetailsFactory is a public interface that we want the API project to call InternalServerError when there is an uncaught exception or BadRequest when there are errors.

We then register the internal ProblemDetailsFactory class to be used whenever IProblemDetailsFactory is injected.

That's right, take that statement in. This project's (Middleware) internal implementation of the public interface is used in a different assembly (API). The factory, as intended, can't be initiated outside of the assembly. The factory is still internal to its project and has access to anything in that assembly (Middleware). It's conceptually like exposing a public method to return an internal instance. Cool.

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

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Tips.Middleware.ErrorHandling;

namespace Tips.Middleware.Configuration

{

public static class DependencyRegistrar

{

public static void Register(IServiceCollection services, IConfiguration configuration)

{

// Middleware must be injected as a singleton.

var problemDetailsConfiguration = new ProblemDetailsConfiguration();

configuration.Bind(nameof(ProblemDetailsConfiguration), problemDetailsConfiguration);

services.AddSingleton(problemDetailsConfiguration);

// This is a dependency within the Middleware class, so it too must be injected as a singleton.

services.AddSingleton(typeof(IProblemDetailsFactory), typeof(ProblemDetailsFactory));

}

}

}

# Tips.Middleware.ErrorHandling.ProblemDetailsConfiguration

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware/ErrorHandling/ProblemDetailsConfiguration.cs>

namespace Tips.Middleware.ErrorHandling

{

internal class ProblemDetailsConfiguration

{

public string UrnName { get; set; }

}

}

# Tips.Middleware.ErrorHandling.IProblemDetailsFactory

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware/ErrorHandling/IProblemDetailsFactory.cs>

using System.Collections.Generic;

using Microsoft.AspNetCore.Mvc;

using Tips.Pipeline;

namespace Tips.Middleware.ErrorHandling

{

public interface IProblemDetailsFactory

{

ProblemDetailsWithNotifications BadRequest(List<Notification> notifications);

ProblemDetails InternalServerError();

}

}

# Tips.Middleware.ErrorHandling.ProblemDetailsFactory

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Middleware/ErrorHandling/ProblemDetailsFactory.cs>

using System.Collections.Generic;

using System.Net;

using Microsoft.AspNetCore.Mvc;

using Tips.Pipeline;

namespace Tips.Middleware.ErrorHandling

{

internal class ProblemDetailsFactory : IProblemDetailsFactory

{

…

private readonly ProblemDetailsConfiguration \_configuration;

public ProblemDetailsFactory(ProblemDetailsConfiguration configuration) => \_configuration = configuration;

public ProblemDetailsWithNotifications BadRequest(List<Notification> notifications)

{  
…

}

public ProblemDetails InternalServerError()

{  
…

}

}

}

# Tips.TodoItems.Configuration.DependencyRegistrar

Building upon what we've already learned, let's check out a few more examples.

Each project can be fully isolated with full control of how it performs CRUD operations. This project adds a database context that it uses. This implementation uses an in-memory database, but we could have injected the configuration, bind it to an internal configuration object, and register an actual database implementation.

Granted, the database context could be used by other projects, but following separation of concerns, each project may have its own database. Of course, if it's truly a shared resource, then just register it in the Startup. For this application, this is the only project using a database, so let's isolate it here to reenforce that ownership.

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/TodoItems/Configuration/DependencyRegistrar.cs#L29>

services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));

The repositories and their interfaces are internal, so they cannot be accessed outside of this assembly.

Example: ICreateTodoItemRepository and CreateTodoItemRepository are both internal.

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/TodoItems/Configuration/DependencyRegistrar.cs#L46>

services.AddScoped(typeof(ICreateTodoItemRepository), typeof(CreateTodoItemRepository));

Here's the relevant part of the class for context.

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/TodoItems/Configuration/DependencyRegistrar.cs>

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.DependencyInjection;

using Tips.Pipeline;

using Tips.Rules;

using Tips.TodoItems.Context;

using Tips.TodoItems.Context.Models;

using Tips.TodoItems.Handlers.CreateTodoItem;

using Tips.TodoItems.Handlers.DeleteTodoItem;

using Tips.TodoItems.Handlers.GetTodoItem;

using Tips.TodoItems.Handlers.GetTodoItems;

using Tips.TodoItems.Handlers.UpdateTodoItem;

using Tips.TodoItems.Models;

using Tips.TodoItems.Rules.CreateRules;

using Tips.TodoItems.Rules.SaveRules;

using Tips.TodoItems.Rules.UpdateRules;

namespace Tips.TodoItems.Configuration

{

public static class DependencyRegistrar

{

public static void Register(IServiceCollection services)

{

services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));

…

services.AddScoped(typeof(ICreateTodoItemRepository), typeof(CreateTodoItemRepository));

services.AddScoped(typeof(IUpdateTodoItemRepository), typeof(UpdateTodoItemRepository));

…

}

}

}

# Tips.TodoItems.Handlers.CreateTodoItem.ICreateTodoItemRepository

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/TodoItems/Handlers/CreateTodoItem/ICreateTodoItemRepository.cs>

using System.Threading.Tasks;

using Tips.Pipeline;

using Tips.TodoItems.Context.Models;

namespace Tips.TodoItems.Handlers.CreateTodoItem

{

internal interface ICreateTodoItemRepository

{

Task SaveAsync(Response<TodoItemEntity> response);

}

}

# Tips.TodoItems.Handlers.CreateTodoItem.CreateTodoItemRepository

<https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/TodoItems/Handlers/CreateTodoItem/CreateTodoItemRepository.cs>

using System.Threading.Tasks;

using Tips.Pipeline;

using Tips.TodoItems.Context;

using Tips.TodoItems.Context.Models;

namespace Tips.TodoItems.Handlers.CreateTodoItem

{

internal class CreateTodoItemRepository : ICreateTodoItemRepository

{

private readonly TodoContext \_context;

public CreateTodoItemRepository(TodoContext context) => \_context = context;

public async Task SaveAsync(Response<TodoItemEntity> response)

{

await \_context.TodoItems.AddAsync(response.Item);

await \_context.SaveChangesAsync();

}

}

}

# References

1. Assembly Encapsulation: [Dependency Injection of Internals in .NET Core 2.1](https://www.danylkoweb.com/Blog/dependency-injection-of-internals-in-net-core-21-OQ)
2. Assembly Encapsulation: [Dependency Injection of Internals, Part 2](https://www.danylkoweb.com/Blog/dependency-injection-of-internals-part-2-HB)

# Conclusion

We've continued the journey of Assembly Encapsulation via dependency injection of internals in C# .NET 5 (Core). In this iteration, we used the built in .NET IoC to register an assembly's dependencies by calling a static method in each project to perform the registration for us. This separates the concerns making it very clear which project initiated the registration. The objects registered take advantage of the access modifiers. Projects expose only the public classes and interfaces that should be accessible. Internal classes and interfaces stay internal and while they are registered, no other assemblies may use them.

Were you disappointed in throwing out Assembly Encapsulation to gain the benefits of Dependency Injection? That you had to choose one or the other? Do the techniques above or written in the two previous articles give you confidence that you can have both at the same time? Share your thoughts below.