2. Common Structures

2.1. Dependency Injection

2.1.1. What is Dependency Injection

If you already know Dependency Injection concept, Constructor and Property Injection patterns, you can skip to the next section.

Wikipedia says: "*Dependency injection is a software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object (or client) and are made part of the client's state. The pattern separates the creation of a client's dependencies from its own behavior, which allows program designs to be loosely coupled and to follow the dependency inversion and single responsibility principles. It directly contrasts the service locator pattern, which allows clients to know about the system they use to find dependencies*."

It's very hard to manage dependencies and develop a modular and well structured application without using dependency injection techniques.

Problems of Traditional Way

In an application, classes depend on each other. Assume that we have an application service that uses a repository to insert entities to database. In this situation, the application service class is dependent to the repository class. See the example:

```
public class PersonAppService
{
    private IPersonRepository personRepository;

    public PersonAppService()
    {
        this.personRepository = new PersonRepository();
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        this.personRepository.Insert(person);
    }
}
```

PersonAppService uses **PersonRepository** to insert a **Person** to the database. Problems of this code:

- PersonAppService uses **IPersonRepository** reference in **CreatePerson** method, so this method depends on IPersonRepository, instead of PersonRepository concrete class. But the PersonAppService still depends on PersonRepository in it's constructor. Components should depend on interfaces rather than implementation. This is known as Dependency Inversion principle.
- If the PersonAppService creates the PersonRepository itself, it become dependent to a specific implementation of IPersonRepository interface and can not work with another implementation. Thus, separating interface from implementation becomes meaningless. Hard-dependencies makes code base tightly coupled and low re-usable.
- We may need to change creation of PersonRepository in the future. Say, we may want to make it singleton (single shared instance rather than creating an object for each usage). Or we may want to create more than one class those implement IPersonRepository and we want to create one of them conditionally. In this situation, we should change all classes that depends on IPersonRepository.
- With such a dependency, it's very hard (or impossible) to unit test the PersonAppService.

To overcome some of these problems, factory pattern can be used. Thus, creation of repository class is abstracted. See the code below:

```
public class PersonAppService
{
    private IPersonRepository personRepository;

    public PersonAppService()
    {
        this.personRepository = PersonRepositoryFactory.Create();
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        this.personRepository.Insert(person);
    }
}
```

PersonRepositoryFactory is a static class that creates and returns an IPersonRepository. This is known as **Service Locator** pattern. Creation problems are solved since PersonAppService does not know how to create

an implementation of IPersonRepository and it's independent from PersonRepository implementation. But, still there are some problems:

- In this time, PersonAppService depends on PersonRepositoryFactory. This is more acceptable but still there is a hard-dependency.
- It's tedious to write a factory class/method for each repository or for each dependency.
- It's not well testable again, since it's hard to make PersonAppService to use some mock implementation of IPersonRepository.

Solution

There are some best practices (patterns) to depend on other classes.

Constructor Injection Pattern

The example above can be re-written as shown below:

```
public class PersonAppService
{
    private IPersonRepository personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        this.personRepository = personRepository;
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        this.personRepository.Insert(person);
    }
}
```

This is known as **constructor injection**. Now, PersonAppService does not know which classes implement IPersonRepository and how to create it. Who needs to use PersonAppService, first creates an IPersonRepository and pass it to constructor of PersonAppService as shown below:

```
var repository = new PersonRepository();
var personService = new PersonAppService(repository);
personService.CreatePerson("Yunus Emre", 19);
```

Constructor Injection is a perfect way of making a class independent to creation of dependent objects. But, there are some problems with the code above:

- Creating a PersonAppService become harder. Think that it has 4 dependency, we must create these 4 dependent object and pass them into the constructor of the PersonAppService.
- Dependent classes may have other dependencies (Here, PersonRepository may has dependencies). So, we must create all dependencies of PersonAppService, all dependencies of dependencies and so on.. In that way, we may not even create a single object since dependency graph is too complex.

Fortunately, there are Dependency Injection frameworks automates to manage dependencies.

Property Injection pattern

Constructor injection pattern is a perfect way of providing dependencies of a class. In this way, you can not create an instance of the class without supplying dependencies. It's also a strong way of explicitly declaring what's the requirements of the class to properly work.

But, in some situations, the class depends on another class but also can work without it. This is usually true for cross-cutting concerns such as logging. A class can work without logging, but it can write logs if you supply a logger. In this case, you can define dependencies as public properties rather than getting them in constructor. Think that we want to write logs in PersonAppService. We can re-write the class as like below:

```
public class PersonAppService
{
    public ILogger Logger { get; set; }
    private IPersonRepository personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        this.personRepository = personRepository;
        Logger = NullLogger.Instance;
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        this.personRepository.Insert(person);
        Logger.Debug("Successfully inserted!");
    }
}
```

NullLogger.Instance is a singleton object that implements ILogger but actually does nothing (does not write logs. It implements ILogger with empty method bodies). So, now, PersonAppService can write logs if you set Logger after creating the PersonAppService object like below:

```
var personService = new PersonAppService(new PersonRepository());
personService.Logger = new Log4NetLogger();
personService.CreatePerson("Yunus Emre", 19);
```

Assume that Log4NetLogger implements ILogger and write logs using Log4Net library. Thus, PersonAppService can actually write logs. If we do not set Logger, it does not write logs. So, we can say the ILogger is an optional dependency of PersonAppService.

Almost all of Dependency Injection frameworks support Property Injection pattern.

Dependency Injection frameworks

There are many dependency injection framework that automates resolving dependencies. They can create objects with all dependencies (and dependencies of dependencies recursively). So, you just write your class with constructor & property injection patterns, DI framework handles the rest! In a good application, your classes are independent even from DI framework. There will be a few lines of code or classes that explicitly interact with DI framework in your whole application.

StudioX uses Castle Windsor framework for Dependency Injection. It's one of the most mature DI frameworks. There are many other frameworks like Unity, Ninject, StructureMap, Autofac and so on.

In a dependency injection framework, you first register your interfaces/classes to the dependency injection framework, then you can resolve (create) an object. In Castle Windsor, it's something like that:

```
var container = new WindsorContainer();
container.Register(
        Component.For<IPersonRepository>().ImplementedBy<PersonRepository>().LifestyleTransient(),
        Component.For<IPersonAppService>().ImplementedBy<PersonAppService>().LifestyleTransient()
    );

var personService = container.Resolve<IPersonAppService>();
personService.CreatePerson("Yunus Emre", 19);
```

We first created the WindsorContainer then registered PersonRepository and PersonAppService with their interfaces. Then we asked to container to create an IPersonAppService. It created PersonAppService with

dependencies and returned back. Maybe it's not so clear of advantage of using a DI framework in this simple example, but think you will have many classes and dependencies in a real enterprise application. Surely, registering dependencies will be somewhere else from creation and using objects and made only one time in an application startup.

Notice that we also declared life cycle of objects as transient. That means whenever we resolves an object of these types, a new instance will be created. There are many different life cycles (like singletion).

2.1.2. StudioX Dependency Injection Infrastructure

StudioX almost makes invisible of using the dependency injection framework when you write your application by following best practices and some conventions.

Registering Dependencies

There are different ways of registering your classes to Dependency Injection system in StudioX. Most of time, conventional registration will be sufficient.

Conventional Registrations

StudioX automatically registers all Repositories, Domain Services, Application Services, MVC Controllers and Web API Controllers by convention. For example, you may have a IPersonAppService interface and a PersonAppService class that implements it:

```
public interface IPersonAppService : IApplicationService
{
}

public class PersonAppService : IPersonAppService
{
}
```

StudioX automatically registers it since it implements **IApplicationService** interface (It's just an empty interface). It's registered as **transient** (created instance per usage). When you inject (using constructor injection) IPersonAppService interface to a class, a PersonAppService object is created and passed into constructor automatically.

Naming conventions are very important here. For example you can change name of PersonAppService to MyPersonAppService or another name which contains 'PersonAppService' postfix since the IPersonAppService has this postfix. But you can not name your service as PeopleService. If you do it, it's not registered for IPersonAppService automatically (It's registered to DI framework but with self-registration, not with interface), so, you should manually register it if you want.

StudioX can register assemblies by convention. You can tell StudioX to register your assembly by convention. It's pretty easy:

```
IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());
```

Assembly.GetExecutingAssembly() gets a reference to the assembly which contains this code. You can pass other assemblies to RegisterAssemblyByConvention method. This is generally done when your module is being initialized. See StudioX's module system for more.

You can write your own conventional registration class by implementing IConventionalRegisterer interface and calling IocManager.AddConventionalRegisterer method with your class. You should add it in pre-initialize method of your module.

Helper Interfaces

You may want to register a specific class that does not fit to conventional registration rules. StudioX provides ITransientDependency and ISingletonDependency interfaces as a shortcut. For example:

```
public interface IPersonManager
{
}

public class MyPersonManager : IPersonManager, ISingletonDependency
{
}
```

In that way, you can easily register MyPersonManager. When need to inject a IPersonManager, MyPersonManager class is used. Notice that dependency is declared as Singleton. Thus, a single instance of MyPersonManager is created and same object is passed to all needed classes. It's just created in first usage, then same instance is used in whole life of the application.

Custom/Direct Registration

If conventional registrations are not sufficient for your situation, you can either use IocManager or Castle Windsor to register your classes and dependencies.

Using IocManager

You can use IocManager to register dependencies (generally in PreInitialize of your module definition class):

```
IocManager.Register<IMyService, MyService>(DependencyLifeStyle.Transient);
```

Using Castle Windsor API

You can use **IIocManager.IocContainer** property to access to the Castle Windsor Container and register dependencies. Example:

```
IocManager.IocContainer.Register(Classes.FromThisAssembly().BasedOn<IMySpecialInterface>().Lifesty
lePerThread().WithServiceSelf());
```

For more information, read Windsor's documentation

Resolving

Registration informs IOC (Inversion of Control) Container (a.k.a. DI framework) about your classes, their dependencies and lifetimes. Somewhere in your application you need to create objects using IOC Container. ASP.NET Provides a few options for resolving dependencies.

Constructor & Property Injection

You can use constructor and property injection to get dependencies to your classes as **best practice**. You should do it that way wherever it's possible. An example:

```
public class PersonAppService
{
    public ILogger Logger { get; set; }
    private IPersonRepository personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        this.personRepository = personRepository;
        Logger = NullLogger.Instance;
```

```
    }

    public void CreatePerson(string name, int age)
    {
        var person = new Person { Name = name, Age = age };
        this.personRepository.Insert(person);
        Logger.Debug("Successfully inserted!");
    }
}
```

IPersonRepository is injected from constructor and ILogger is injected with public property. In that way, your code will be unaware of dependency injection system at all. This is the most proper way of using DI system.

IIocResolver, IIocManager and IScopedIocResolver

You may have to directly resolve your dependency instead of constructor & property injection. This should be avoided when possible, but it may be impossible. StudioX provides some services those can be injected and used easily. Example:

```
public class MySampleClass : ITransientDependency
{
    private readonly IIocResolver iocResolver;
    public MySampleClass(IIocResolver iocResolver){ this.iocResolver = iocResolver; }

    public void DoIt()
    {
        //Resolving, using and releasing manually
        var personService1 = this.iocResolver.Resolve<PersonAppService>();
        personService1.CreatePerson(new CreatePersonInput { Name = "Yunus"});
        this.iocResolver.Release(personService1);

        //Resolving and using in a safe way
        using (var personService2 = this.iocResolver.ResolveAsDisposable<PersonAppService>())
        {
            personService2.Object.CreatePerson(new CreatePersonInput { Name = "Yunus"});
        }
    }
}
```

MySampleClass in an example class in an application. It constructor-injected **IIcResolver** and used it to resolve and release objects. There are a few overloads of **Resolve** method can be used as needed. **Release** method is used to release component (object). It's **critical** to call Release if you're manually resolving an object. Otherwise, your application may have memory leak problem. To be sure of releasing the object, use

ResolveAsDisposable (as shown in the example above) wherever it's possible. It automatically calls Release at the end of the using block.

IIocResolver (and IIocManager) have also **CreateScope** extension method (defined in StudioX.Dependency namespace) to safely release all resolved dependencies. Example:

```
using (var scope = iocResolver.CreateScope())
{
    var simpleObj1 = scope.Resolve<SimpleService1>();
    var simpleObj2 = scope.Resolve<SimpleService2>();

    //...
}
```

In the end of using block, all resolved dependencies automatically removed. A scope is also injectable using **IScopedIocResolver**. You can inject this interface and resolve dependencies. When your class is released, all resolved dependencies will be released. But use it carefully; For example, if your class has a long life (say it's a singleton) and you are resolving too many objects, then all of them will remain in the memory until your class is released.

If you want to directly reach to IOC Container (Castle Windsor) to resolve dependencies, you can consturctor-inject **IIocManager** and use **IIocManager**.**IocContainer** property. If you are in a static context or can not inject IIocManager, as a last chance, you can use singleton object **IocManager**.**Instance** everywhere. But, in that case your codes will not be easy to test.

Extras

IShouldInitialize interface

Some classes need to be initialized before first usage. IShouldInitialize has an Initialize() method. If you implement it, then your Initialize() method is automatically called just after creating your object (before used). Surely, you should inject/resolve the object in order to work this feature.

ASP.NET MVC & ASP.NET Web API integration

We must call dependency injection system to resolve the root object in dependency graph. In an **ASP.NET MVC** application, it's generally a **Controller** class. We can use contructor injection and property injection

patterns also in controllers. When a request come to our application, the controller is created using IOC container and all dependencies are resolved recursively. So, who does that? It's done automatically by StudioX by extending ASP.NET MVC's default controller factory. As similar, it's true for ASP.NET Web API also. You don't care about creating and disposing objects.

ASP.NET Core Integration: ASP.NET Core has already a built-in dependency injection system with Microsoft.Extensions.DependencyInjection package. StudioX uses Castle.Windsor.MsDependencyInjection package to integrate it's dependency injection system with ASP.NET Core's. So, you don't have to think about it.

Last notes

StudioX simplifies and automates using dependency injection as long as you follow the rules and use the structures above. Most of times you will not need more. But if you need, you can directly use all power of Castle Windsor to perform any task (like custom registrations, injection hooks, interceptors and so on).