Dependency Injection in practice

Kristijan Horvat
Software Architect
twitter.com/khorvat2

What is Dependency Injection (DI)

Dependency injection is a software design pattern that implements inversion of control for software libraries.

Wikipedia

Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.

Mark Seemann

What is Inversion of Control (IoC)

Inversion of Control (IoC) describes a design in which custom-written portions of a computer program receive the flow of **control** from a generic, reusable library.

Wikipedia

Inversion of Control is a common pattern among developers that helps assemble components from different projects into a cohesive application.

~ *based on* http://www.martinfowler.com/articles/injection.html *(reworded)*.

Martin Fowler

When to use DI & IoC ?

High Level perspective

- when your software stack supports DI & IoC
- when you have mid to large projects or teams
- when we have long running projects
- when your project tend to change over time (Agile)
- when you plan to use unit testing
- when you plan to deploy projects to cloud (Amazon, Azure, etc.)

When to use DI & IoC ?

Low Level perspective

- when we want to avoid tight coupling of objects
- when we need to easily add new functionality to code (Extensibility)
- when we need to choose what components we use at run-time rather than compile-time (Late binding)
- when we want multiple development teams to work on the same project (Parallel Development)
- when we want independent components and isolated functionality (Maintainability)
- when we need to use mocks for unit testing (Testability)
- when we want to enforce SRP (single responsibility principle)

DI Containers

.NET
- AutoFac
- SimpleInjector
- Ninject
- StructureMap
- Castle Windsor
- Unity
- Spring.NET

PHP
- Laravel IoC
- PHP DI
- Zend DI
- Symfony
- Dice

Java
- Pico container
- Guice
- Spring
- Silk DI

JavaScript
- di-lite
- inverted
- wire.js
- bottle.js
- pimple
- cujo.js (Spring like)

How to choose DI container ?

Things to consider while choosing DI container

- easy to understand API
- easy and readable configuration
- performance
- plugin support
- container should be widely accepted
- extensions (the more the merrier)
- large community (maybe the most important)

Tag: ##

DI in Practice - GitHub Repo

1. Prerequisites
    1. .Net 4.x
    2. VS 2013
    3. Git
2. Repository setup
    1. git clone https://github.com/khorvat/DependencyInjectionInPractice
    2. git checkout master
    3. git checkout tags/## (git checkout tags/01)

Note: Slides that have Tag marker in the top right corner follows the GitHub code samples.

Tag: 01

So how can we use DI ?

Dependency Injection By Hand

Dependency

```
class Sword
{
    public void Hit(string target)
    {
        Console.WriteLine("Chopped {0} clean in half", target);
    }
}
```

Note

- tight coupling
- hard to mock
- missing abstraction

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

Tag: 01

So how can we use DI ?

Dependency Injection By Hand

Target

```
class Samurai
{
    readonly Sword sword;
    public Samurai()
    {
        this.sword = new Sword();
    }
...
```

```
...
 public void Attack(string target)
    {
        this.sword.Hit(target);
    }
}
```

Note

- tight coupling
- hard to mock
- missing abstraction

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

So how can we use DI ?

Dependency Injection By Hand

Create Target

```
class Program
{
    public static void Main()
    {
        var warrior = new Samurai();
        warrior.Attack("the evildoers");
    }
}
```

Note

- unable to inject different implementation

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

Tag: 02

To decouple we need to abstract ?

Things we need to do in order to use DI and IoC

- program with interfaces or abstract classes
- hosting class needs to use interfaces
    - can receive any implementation
    - can have real one for production
    - can have mocked one for unit testing
- expose dependencies via constructor (constructor injection)

Abstraction

```
interface IWeapon
{
    void Hit(string target);
}
```

So how can we use DI ?

Dependency Injection By Hand - Abstraction

Dependency

```
class Sword : IWeapon
{
  public void Hit(string target)
  {
    Console.WriteLine("Chopped {0} clean in half", target);
  }
}
```

```
class Shuriken : IWeapon
{
  public void Hit(string target)
  {
    Console.WriteLine("Pierced {0}'s armor", target);
  }
}
```

Note

- loosely coupled
- IoC support
- easy to mock

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

So how can we use DI ?

Dependency Injection By Hand - Abstraction

Target

```
class Samurai
{
    readonly IWeapon weapon;
    public Samurai(IWeapon weapon)
    {
        this.weapon = weapon;
    }
...
```

```
...
    public void Attack(string target)
    {
        this.weapon.Hit(target);
    }
}
```

Note

- loosely coupled
- IoC support
- easy to mock

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

Tag: 03

So how can we use DI ?

Dependency Injection By Hand - Abstraction

Create Target

```
class Program
{
    public static void Main()
    {
        var warrior1 = new Samurai(new Shuriken());
        var warrior2 = new Samurai(new Sword());
        warrior1.Attack("the evildoers");
        warrior2.Attack("the evildoers");
    }
}
```

The following results will be printer to console output:

Pierced the evildoers armor.
Chopped the evildoers clean in half.

Examples from Ninject - https://github.com/ninject/Ninject/wiki/

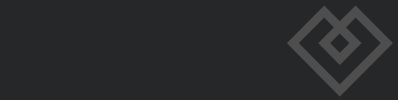
Tag: 03

So how can we use DI ?

Dependency Injection By Hand - Abstraction

- we have loosely coupling
- we have abstractions
- we have IoC support
- we have constructor injection
- we have dependency injection – by hand

- what happens when our dependencies has dependencies on their own ?



Examples from Ninject - https://github.com/ninject/Ninject/wiki/

Tag: 04

Great, but do I have to resolve every dependency by hand ?

DI Container to rescue!

```
IKernel kernel = new StandardKernel();
var samurai = kernel.Get<Samurai>();
```

Tag: 05

Ok, this works for concrete implementation, what about interfaces?

```
IKernel kernel = new StandardKernel();
var warrior = kernel.Get<IWarrior>();
```

We need DI container configuration.

Configuration Practices – Code, XML or Convention ?

Code

- strongly-typed
- refactoring and compiler support
- easy to maintain
- conditional bindings support

Note: You can mix configurations

Configuration Practices – Code, XML or Convention ?

XML

- hot-swap (no recompilation)
- no refactoring or compiler support
- verbose and hard to maintain
- hard to enforce conditional bindings

Note: You can mix configurations

Configuration Practices – Code, XML or Convention ?

Convention

- hot-swap
- conditional bindings support
- hard to maintain
- great deal of magic is involved

Note: You can mix configurations

Configuration using Code

```
IKernel kernel = new StandardKernel();

kernel.Bind<IWeapon>().To<Shuriken>();
kernel.Bind<IWarrior>().To<Samurai>();

var warrior = kernel.Get<IWarrior>();
```

Code

- strongly-typed
- refactoring and compiler support
- easy to maintain
- conditional bindings support

Patterns and Practices

- Injection Patterns
- Multi Injection
- Abstract Factory Pattern
- Facade Services (or Aggregate Service)
- Composition Root

Injection Patterns - Constructor Injection

Constructor Injection

public (IWeaponAction weaponAction)

- clean implementation
- not bound to specific DI container
- easy construct tests

Injection Patterns - Property Injection

Property Injection

[Inject]

public IWeapon Weapon {get; set; }

- hides implementation
- bound to specific DI container
- exposing internal architecture

Injection Patterns – Method Injection

Method Injection

[Inject]

public void Arm(IWeapon weapon)

- hides implementation
- bound to specific DI container
- initialization logic needed

Tag: 06

Multi Injection

Inject multiple objects bound to a particular type or interface

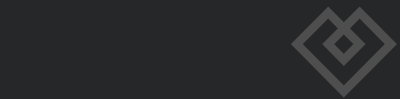
```
IKernel kernel = new StandardKernel();

kernel.Bind<IWeapon>().To<Shuriken>()
;
kernel.Bind<IWeapon>().To<Sword>();
kernel.Bind<IWarrior>().To<Samurai>();

var warrior = kernel.Get<IWarrior>();
```

```
public Samurai(List<IWeapon> weapons)
{
        this.weapons = weapons;
}
```

Tag: 07

Abstract Factory Pattern

- Abstract Factory is design pattern where an interface is responsible for creating related objects without explicitly specifying their concrete classes
- lightweight implementation (or constructed on runtime via Dynamic Proxy)
- used for injecting optional services (gain performance by reducing resolution time)
- The *new* operator is considered harmful (no IoC)



Tag: 07

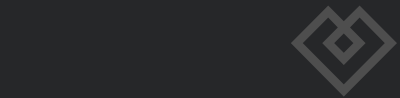
Abstract Factory Pattern

```
public interface IDaggerFactory
{
    IDagger Create();
}
....
public Samurai(IDaggerFactory daggerFactory)
....
daggerFactory.Create().Hit(target);
....
```

Tag: 08

Facade Services (or Aggregate Services) Concept

- Factory Service is a design concept where an interface is used to aggregate any number of services or factories to overcome the constructor over-injection and possible performance implications
- using more than 3-4 services in the constructor is a clear sign that we should consider using facade service
- using constructor injection makes it easy to determine what services should be aggregated



Tag: 08

Facade Services (or Aggregate Services) Concept

```
public interface IWeaponFactory
{
    IDagger CreateDagger();
    ISword CreateSword();
}
....
public Samurai(IWeaponFactory weaponFactory)
....
weaponFactory.CreateDagger().Hit(target);
weaponFactory.CreateSword().Hit(target);
....
```

Composition Root

- Composition Root is a (preferably) unique location in an application where modules are composed together. (M. Seemann)
- only composition root should have reference to DI container
- code relies on injection patterns but is never composed
- only applications should have Composition Roots, libraries and frameworks shouldn't.
- entire object graph should be composed in the following entry points (depends on the framework)
  - console application – Main method
  - ASP.NET MVC & WebAPI applications - global.asax, IControllerFactory or PreApplicationStartMethod
  - etc.

Composition Root

WebAPI Example

```
private static readonly Bootstrapper bootstrapper = new Bootstrapper();
....
bootstrapper.Initialize(CreateKernel);
....
private static IKernel CreateKernel()
{
        var kernel = new StandardKernel();
        // Install Ninject-based IDependencyResolver into the Web API configuration set Web API Resolver
        GlobalConfiguration.Configuration.DependencyResolver = new NinjectDependencyResolver(kernel);
        return kernel;
}
```

Tag: 09,10,11

What about memory management ?

- Some of the DI containers manage object lifecycle automatically and some have object scopes
- It's important to know how you objects are managed, with Ninject DI there are following scopes available
  - Transient – not managed by the Kernel – no scope
  - Singleton – objects are disposed when Kernel is disposed
  - Thread - objects are disposed when underlying Thread object is garbage collected.
  - Request – Web Request - objects are disposed at the end of the Web request processing
  - Named, Call & Parent - objects are disposed when their scope object is GC'd
  - Custom Scope – you manage object lifecycle

Good and Bad Practices

- Good
  - Extensibility
  - Dynamic Proxy
  - IoC of DI Container
- Bad
  - Constructor Over-Injection
  - Service Locator
  - Non-Abstract Factories

Tag: 12

Extensibility - Good Practices

- Work on another implementation in parallel
- Switch implementations dynamically
- Enforce SRP
- Produce clean and maintainable projects

Tag: 13

Dynamic Proxy - Good Practices

- Abstract Factories and Facade Services can be constructed on the runtime by using the dynamically generated proxy classes
- Classes are created on application startup (only once) and loaded into memory (AppDomain)
- Pros
  - easy to implement
  - lightweight – no code at all
- Cons
  - no ready-to-use implementation
  - difficult to understand

Tag: 13

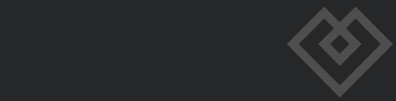
Dynamic Proxy - Good Practices

Usage

```
public interface IDaggerFactory
{
    IDagger Create();
}
....
kernel.Bind<IDaggerFactory>().ToFactory();
....
```

Abstract Away the DI Container - Good Practices

- It's good thing not to be bound to a specific DI container so use abstraction in order to switch from one container to another
  - abstract the container
  - abstract the configuration



Constructor Over-Injection - Bad Practices

- It's a code smell rather then anti-pattern
  - hard to maintain
  - slow resolution
  - resolution of optional dependencies
  - easy detect SRP violation

- **Solution – Facade Services**

Service Locator Anti-Pattern - Bad Practices

- Service Locator is central registry used to obtain services
  - *Kernel.Get<IService>()*
- Service Locator is Anti-Pattern because it hides class dependencies, causing run-time errors rather then compile-time errors – M. Seemann
  - run-time errors
  - easy introduce breaking changes
  - bound to specific DI container
  - it violates SOLID principles – ISP principle

- **Solution – Use Constructor Injection, Factories & Composition Root**

Non-Abstract Factory - Bad Practice

- Non-Abstract Factory is bad practice because you bound factory to concrete implementation

Non-Abstract Factory - Bad Practice

Usage

```
public interface IDaggerFactory
{
        IDagger Create(IMyDependency dep);
}

public class Dagger : IDagger
{
        public Dagger(IMyDependency dep)
        {
        }
}
```

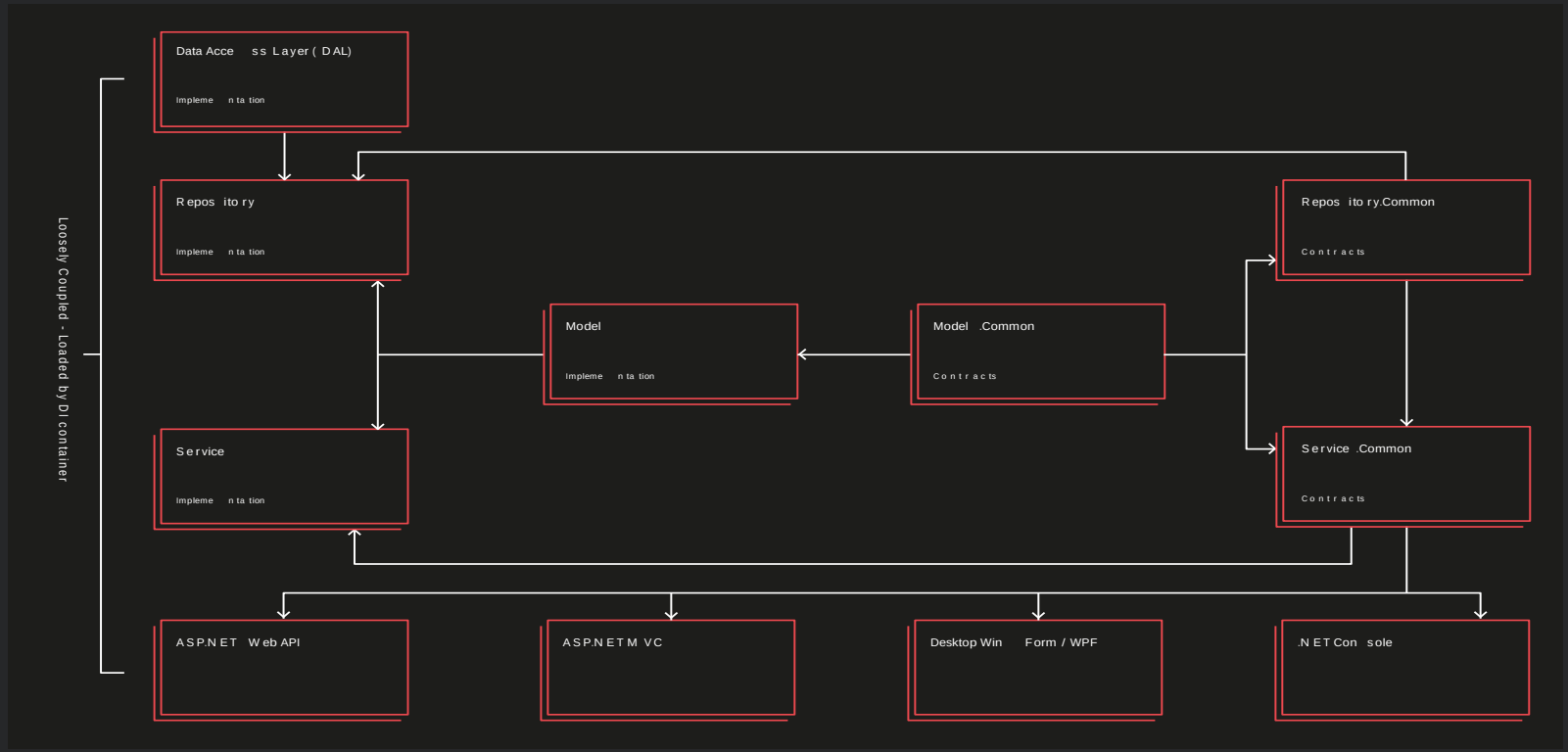
Tag: ##

Layered Architecture - GitHub Repo

1. Prerequisites
    1. .Net 4.x
    2. VS 2013
    3. Git
2. Repository setup
    1. git clone https://github.com/khorvat/DIPracticeLayeredArchitecture
    2. git checkout master
    3. git checkout tags/## (git checkout tags/01)
3. How to Run
    1. Build Solution
    2. Setup IIS
    3. Open Command Prompt in repository root and run RunSample.cmd
        1. Note: In case of IIS Express you will need to edit the RunSample and change the URL of the app

Note: Slides that have Tag marker in the top right corner follows the GitHub code samples.

Layered Architecture with DI and IoC

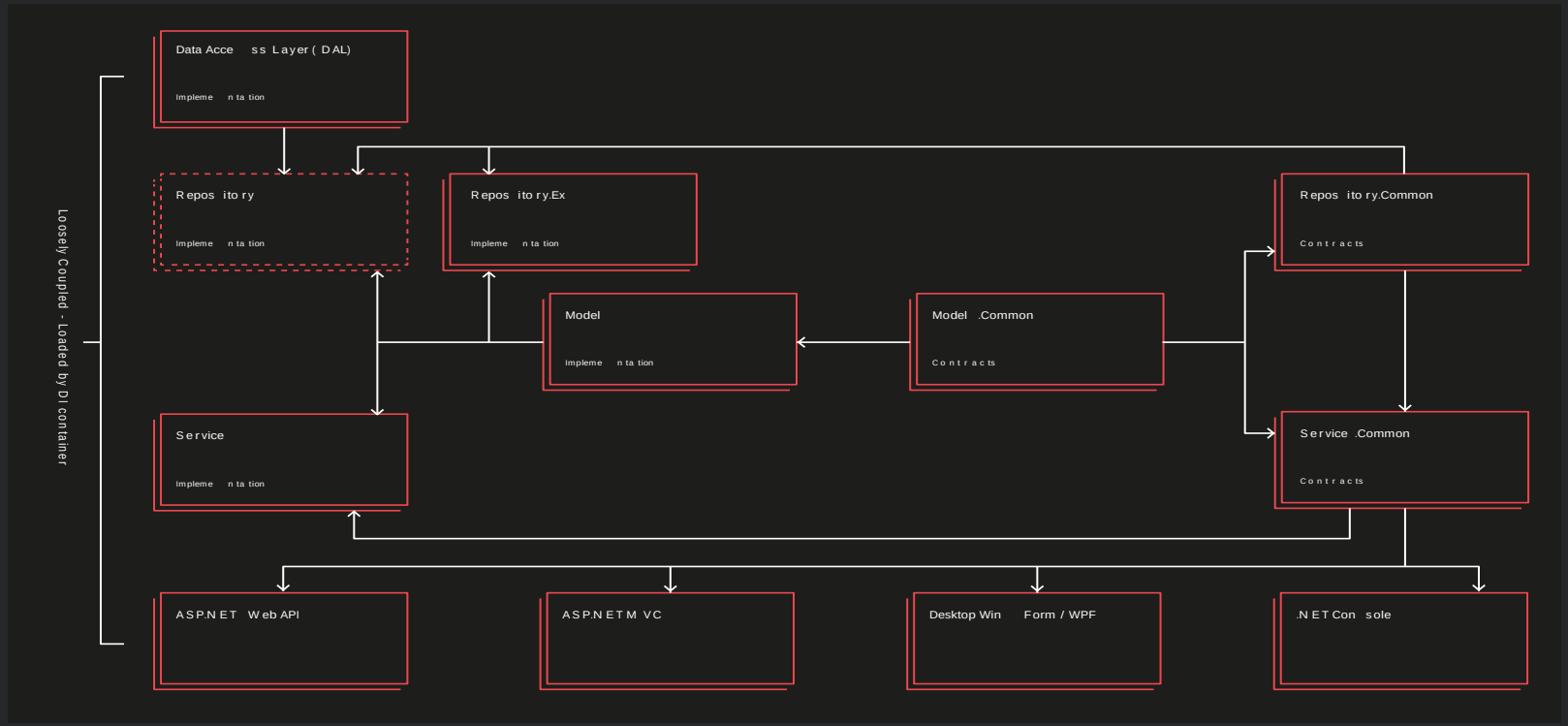


Tag: 01

Cart - Layered Architecture

1. Cart.DAL
    1. CartEntity
    2. ProductEntity
2. Cart.Repository
    1. Get Cart
    2. Get Products
    3. Add Product to Cart
    4. Remove Product from Cart
3. Cart.Service
    1. Get My Cart
    2. Get Only Products InStock
    3. Add Product to Cart With InStock Validation
    4. Remove Product from Cart

Layered Architecture – IoC of the Repository

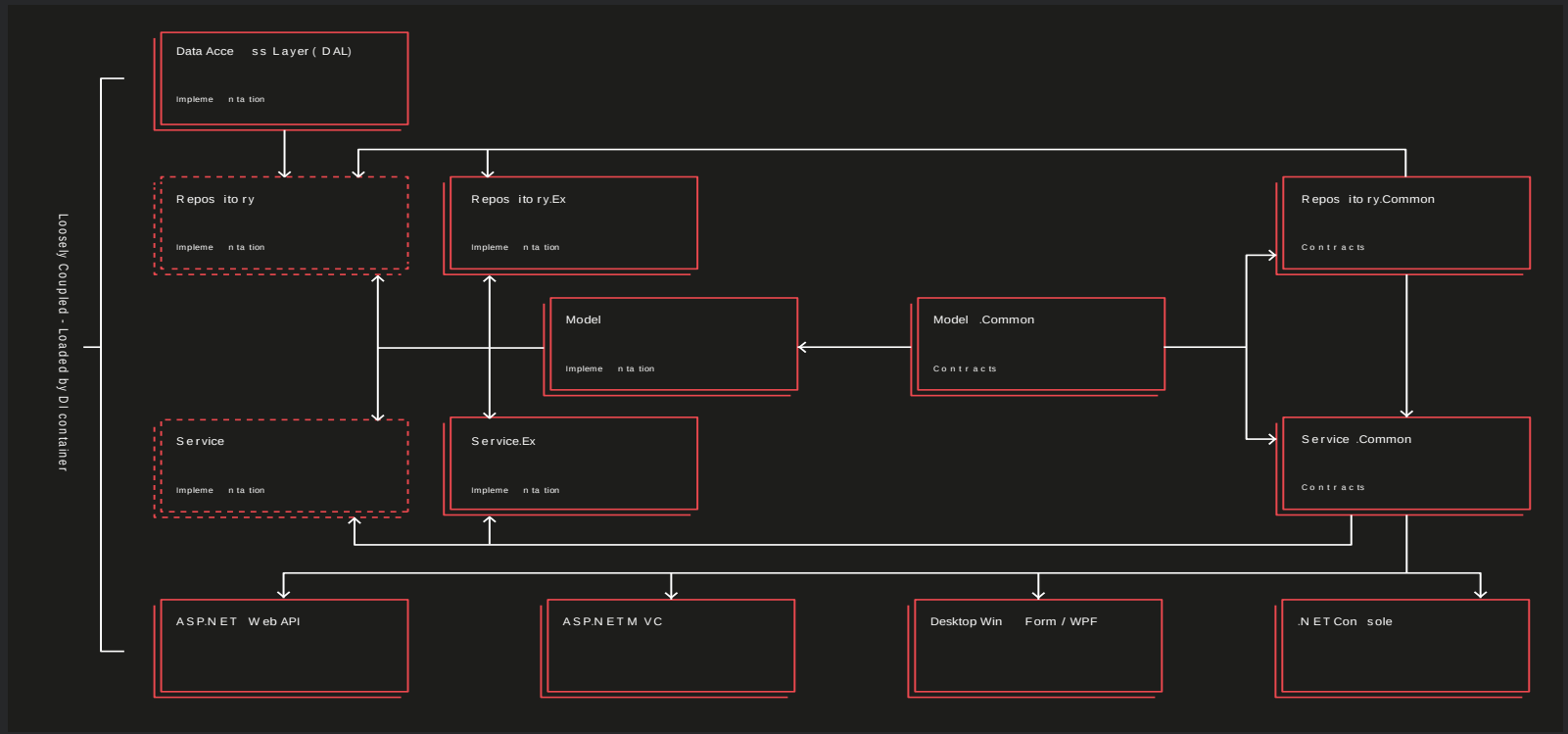


Tag: 02

Cart - IoC of the Repository

1. Cart.DAL
    1. CartEntity
    2. ProductEntity
2. Cart.Repository switched with Cart.Repository.Ex
    1. Get Cart
    2. Get Products With IsDeleted Filter
    3. Add Product to Cart
    4. Remove Product from Cart
3. Cart.Service
    1. Get My Cart
    2. Get Only Products InStock
    3. Add Product to Cart With InStock Validation
    4. Remove Product from Cart

Layered Architecture – IoC of the Service



Tag: 03

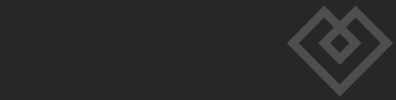
Cart - IoC of the Service

1. Cart.DAL
    1. CartEntity
    2. ProductEntity
2. Cart.Repository switched with Cart.Repository.Ex
    1. Get Cart
    2. Get Products With IsDeleted Filter
    3. Add Product to Cart
    4. Remove Product from Cart
3. Cart.Service switched with Cart.Service.Ex
    1. Get My Cart
    2. Get Only Products InStock with Valid Exp. Date
    3. Add Product to Cart With InStock and Exp. Date Validation
    4. Remove Product from Cart

Cart - Layered Architecture

What have we demonstrated

1. How to setup Ninject DI container inside the ASP.NET WebAPI
2. How should we architecture the layers in order to make them IoC ready
3. We can simply change the Layers as they are loosely coupled



Cart - Layered Architecture

What are practical use cases for this architecture

1. Have one team maintaining existing code while others are working on the new implementation
2. Implement Mocks for whole layers for Unit testing
3. Switch DAL or ORM tool used to access the database
4. Switch file system providers (Local file system to Azure or Amazon storage)
5. Switch caching providers (InMemory Cache to Redis Cache)

Thanks

Questions?

- Kristijan Horvat
- kristijan@mono-software.com
- https://twitter.com/khorvat2

References

- https://github.com/khorvat/DependencyInjectionInPractice
- https://github.com/khorvat/DIPracticeLayeredArchitecture

- http://www.ninject.org/
- https://github.com/ninject/ninject
- http://lukewickstead.wordpress.com/2013/01/18/ninject-cheat-sheet/
- http://www.jeremybytes.com/Downloads/DependencyInjection.pdf
- http://blog.ploeh.dk/
  - http://blog.ploeh.dk/2011/07/28/CompositionRoot/
  - http://blog.ploeh.dk/2012/03/15/ImplementinganAbstractFactory/
  - http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/
  - http://blog.ploeh.dk/2014/05/15/service-locator-violates-solid/
  - http://blog.ploeh.dk/2010/02/02/RefactoringtoAggregateServices/
- http://www.planetgeek.ch/2011/12/31/ninject-extensions-factory-introduction/
- http://www.martinfowler.com/articles/injection.html