## Урок 2

Цель урока: Изучение DI (Dependency Injection). Пример на Ninject и Unity (Autofac, Winsor).

Во многих случаях один и тот же экземпляр типа используется в вашем приложении. Распространенный способ для достижения этой цели является реализация шаблона Одиночка (Singleton).

Проблема в том, что вы в основном должны заблокировать ваш тип. Поступая так, ваш код в конечном итоге будет жестким и негибким. В частности, это будет очень трудно для модульного тестирования кода, который реализован таким образом.

DI контейнер позволяет использовать объекты проинициализированного класса без единого объявления создания объекта в коде.

Первая задача, которую решает DI – это передача за ответственность доступа к сервису от объекта, к «сервису-локатору», который выдает ссылку к сервису.

Например, до использования сервиса:

Опишем пару классов, интерфейс IWeapon с методом Kill, два класса реализации Bazuka и Sword, и класс Warrior, который пользуется оружием:

public interface IWeapon

{

void Kill();

}

public class Bazuka : IWeapon

{

public void Kill()

{

Console.WriteLine("BIG BADABUM!");

}

}

public class Sword : IWeapon

{

public void Kill()

{

Console.WriteLine("Chuk-chuck");

}

}

public class Warrior

{

readonly IWeapon Weapon;

public Warrior(IWeapon weapon)

{

this.Weapon = weapon;

}

public void Kill()

{

Weapon.Kill();

}

}

Используем это:

class Program

{

static void Main(string[] args)

{

Warrior warrior = new Warrior(new Bazuka());

warrior.Kill();

Console.ReadLine();

}

}

Читаем между строк. Создаем воина и даем ему базуку, он идет и убивает. В консоли получаем:

BIG BADABUM!

Заметим, что у нас нету проверки на null в строке

Weapon.Kill();

Что тут плохо? Воин не знает, есть ли у него оружие, и выдачей оружия занимается не отдельный модуль, а главная программа.

Суть DI – поручить выдачу оружия другому модулю.

Подключаем Ninject:

Install-Package Ninject

Создаем модуль, который занимается выдачей оружия:

public class WeaponNinjectModule : NinjectModule

{

public override void Load()

{

this.Bind<IWeapon>().To<Sword>();

}

}

Что буквально значит: «если попросят оружие – то выдайте мечи».

Создаем «сервис-локатор» и пользуемся оружием:

class Program

{

public static IKernel AppKernel;

static void Main(string[] args)

{

AppKernel = new StandardKernel(new WeaponNinjectModule());

var warrior = AppKernel.Get<Warrior>();

warrior.Kill();

Console.ReadLine();

}

}

Как видно объект warrior мы создаем не с помощью конструкции new, а через AppKernel.Get<>(). При создании AppKernel, мы передаем в качестве конструктора модуль, отвечающий за выдачу оружия (в данном случае это меч). Любой объект, который мы пытаемся получить через AppKernel.Get, будет (по мере возможности) проинициализирован, если существуют модули, которые знают, как это делать.

Другой момент применения, когда объект Warrior не берет с собой оружие каждый раз, а при не обнаружении оного обращается к сервису локатору и получает его:

public class OtherWarrior

{

private IWeapon \_weapon;

public IWeapon Weapon

{

get

{

if (\_weapon == null)

{

\_weapon = Program.AppKernel.Get<IWeapon>();

}

return \_weapon;

}

}

public void Kill()

{

Weapon.Kill();

}

}

Исполняем:

var otherWarrior = new OtherWarrior();

otherWarrior.Kill();

Наш воин получает оружие по прямым поставкам – супер!

Ок, в Ninject есть еще одна очень хорошая деталь. Если свойство (public property) помечено [Inject] – то при создании класса через AppKernel.Get<>() –поле инициализуется сервисом-локатором:

public class AnotherWarrior

{

[Inject]

public IWeapon Weapon { get; set; }

public void Kill()

{

Weapon.Kill();

}

}

var anotherWarrior = AppKernel.Get<AnotherWarrior>();

anotherWarrior.Kill();

### Unity

Абсолютно всё то же:

* Установка

Install-Package Unity

* Инициализация сервиса локатора (Container)

Container = new UnityContainer();

* Регистрация типа

Container.RegisterType(typeof(IWeapon), typeof(Bazuka));

* Получение объекта и использование:

var warrior = Container.Resolve<Warrior>();

warrior.Kill();

* Кроме того у Unity есть класс-одиночка (Singleton) ServiceLocator, который регистрирует контейнер и позволяет получить доступ к сервисам из любого места.

var serviceProvider = new UnityServiceLocator(Container);

ServiceLocator.SetLocatorProvider(() => serviceProvider);

* Хитрый OtherWarrior теперь так получает оружие:

public class OtherWarrior

{

private IWeapon \_weapon;

public IWeapon Weapon

{

get

{

if (\_weapon == null)

{

\_weapon = ServiceLocator.Current.GetInstance<IWeapon>();

}

return \_weapon;

}

}

public void Kill()

{

Weapon.Kill();

}

}

### Autofac

Такая же собственно всё и происходит:

* Установка

Install-Package Autofac

* Инициализация строителя сервиса-локатора (ContainerBuilder) – нет-нет, это еще не сам контейнер, это как модули

var builder = new ContainerBuilder();

* Регистрация типов, надо зарегистрировать всех, потому что самостоятельная попытка вызова конструктора без параметров неизвестных типов тут не учитывается

builder.RegisterType<Bazuka>();

builder.RegisterType<Warrior>();

builder.Register<IWeapon>(x => x.Resolve<Bazuka>());

* Создание сервиса локатора (Container)

var container = builder.Build();

* Получение объекта и использование:

var warrior = container.Resolve<Warrior>();

warrior.Kill();

### Castle Windsor

* Установка

Install-Package Castle.Windsor

* Инициализация сервиса-локатора

var container = new WindsorContainer();

* Регистрация типов, надо зарегистрировать всех, потому что самостоятельная попытка вызова конструктора без параметров неизвестных типов тут не учитывается

container.Register(Component.For<IWeapon>().ImplementedBy<Bazuka>(),

Component.For<Warrior>().ImplementedBy<Warrior>());

* Получение объекта и использование:

var warrior = container.Resolve<Warrior>();

warrior.Kill();

### Маленький подытог:

На самом деле реализации Dependency Injection не сильно, но всё же отличаются. Некоторые поддерживают инициализацию в Web.config (App.config) файлах. Некоторые задают правила для инициализации, как мы сейчас посмотрим на расширении Ninject для asp.net mvc – это касается инициализации сервиса-локатора как генератора общих объектов, так и отдельно для каждой сессии.

### Объекты областей (Ninject)

В Ninject можно задать несколько способов инициализации получения объекта из класса. Это требуется для того, что если мы работаем в различных контекстах (например, в разных потоках (Thread)), то объекты должны быть использованы разные. Тем самым поддерживается масштабируемость и гибкость приложения

|  |  |  |
| --- | --- | --- |
| Область | Метод связывания | Объяснение |
| Временный | .InTransientScope() | Объект класса будет создаваться по каждой просьбе. (метод по умолчанию) |
| Одиночка | .InSingletonScope() | Объект класса будет создан однажды и возвращаться будет он по каждому запросу. |
| Поток | .InThreadScope() | Один объект на поток |
| Запрос | .InRequestScope() | Один объект будет создан на каждый web-запрос |

Для нашего web приложения мы будем использовать InRequestScope() для работы с репозиторием (БД).

### Lifetime Manager в Unity

В Unity для задачи правил инициализации используется реализация абстрактного класса LifetimeManager.

Происходит это таким образом:

\_container.RegisterType<DbContext, SavecashTravelContext>(new PerRequestLifetimeManager());

Где PerRequestLifetimeManager – это реализация LifetimeManager:

public class PerRequestLifetimeManager : LifetimeManager

{

/// <summary>

/// Key to store data

/// </summary>

private readonly string \_key = String.Format("SingletonPerRequest{0}", Guid.NewGuid());

/// <summary>

/// Retrieve a value from the backing store associated with this Lifetime policy.

/// </summary>

/// <returns>

/// the object desired, or null if no such object is currently stored.

/// </returns>

public override object GetValue()

{

if (HttpContext.Current != null && HttpContext.Current.Items.Contains(\_key))

return HttpContext.Current.Items[\_key];

return null;

}

/// <summary>

/// Stores the given value into backing store for retrieval later.

/// </summary>

/// <param name="newValue">The object being stored.</param>

public override void SetValue(object newValue)

{

if (HttpContext.Current != null)

HttpContext.Current.Items[\_key] = newValue;

}

/// <summary>

/// Remove the given object from backing store.

/// </summary>

public override void RemoveValue()

{

if (HttpContext.Current != null && HttpContext.Current.Items.Contains(\_key))

HttpContext.Current.Items.Remove(\_key);

}

}

Суть заключается в том, что все объект сохраняется в HttpContext.Current.Items[\_key] и выдается только, если находится в том же контексте HttpContext.Current, иначе создается новый объект. Если HttpContext.Current в области кода не существует (используем такой LifetimeManager в консольном приложении или в отдельном потоке) – то данный контейнер не будет работать.

### Использование Ninject в asp.net mvc

Устанавливаем Ninject в среду asp.net mvc. Отдельно создаем свой проект LessonProject, создадим там HomeController с методом и view Index.

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

}

И

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>LessonProject</h2>

Запустим – работает.

Примечание: В дальнейшем мы будем переносить этот проект в последующие уроки. Так что он так называется (по общему).

Теперь установим модуль Ninject и Ninject.MVC3 для этого проекта.

Install-Package Ninject.MVC3

Добавляется класс в папку App\_Start:

[assembly: WebActivator.PreApplicationStartMethod(typeof(LessonProject.App\_Start.NinjectWebCommon), "Start")]

[assembly: WebActivator.ApplicationShutdownMethodAttribute(typeof(LessonProject.App\_Start.NinjectWebCommon), "Stop")]

namespace LessonProject.App\_Start

{

using System;

using System.Web;

using Microsoft.Web.Infrastructure.DynamicModuleHelper;

using Ninject;

using Ninject.Web.Common;

public static class NinjectWebCommon

{

private static readonly Bootstrapper bootstrapper = new Bootstrapper();

/// <summary>

/// Starts the application

/// </summary>

public static void Start()

{

DynamicModuleUtility.RegisterModule(typeof(OnePerRequestHttpModule));

DynamicModuleUtility.RegisterModule(typeof(NinjectHttpModule));

bootstrapper.Initialize(CreateKernel);

}

/// <summary>

/// Stops the application.

/// </summary>

public static void Stop()

{

bootstrapper.ShutDown();

}

/// <summary>

/// Creates the kernel that will manage your application.

/// </summary>

/// <returns>The created kernel.</returns>

private static IKernel CreateKernel()

{

var kernel = new StandardKernel();

kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);

kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

RegisterServices(kernel);

return kernel;

}

/// <summary>

/// Load your modules or register your services here!

/// </summary>

/// <param name="kernel">The kernel.</param>

private static void RegisterServices(IKernel kernel)

{

}

}

}

В RegisterServices мы добавим инициализации своих сервисов. Для начала добавим шутливый IWeapon, а в дальнейшем еще будем возвращаться к этому методу для регистрации других сервисов:

public interface IWeapon

{

string Kill();

}

…

public class Bazuka : IWeapon

{

public string Kill()

{

return "BIG BADABUM!";

}

}

…

private static void RegisterServices(IKernel kernel)

{

kernel.Bind<IWeapon>().To<Bazuka>();

}

В контроллере используем атрибут [Inject]:

public class HomeController : Controller

{

[Inject]

public IWeapon weapon { get; set; }

public ActionResult Index()

{

return View(weapon);

}

}

Изменяем View:

@model LessonProject.Models.IWeapon

@{

ViewBag.Title = "LessonProject";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

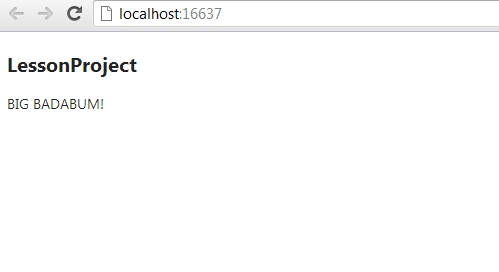
<h2>LessonProject</h2>

<p>

@Model.Kill()

</p>

На выходе получаем:



Ninject пользуясь WebActivator регистрирует свои модули OnePerRequestHttpModule и NinjectHttpModule, а также создает StandartKernel, далее мы можем проинициализировать свои сервисы.

### DependencyResolver

В asp.net mvc 3 появился класс DependencyResolver, который оборачивает DI-контейнеры (но только если сами DI-контейнеры об этом беспокоятся) и его можно использовать как сервис-локатор. Перепишем HomeController:

public class HomeController : Controller

{

private IWeapon weapon { get; set; }

public HomeController()

{

weapon = DependencyResolver.Current.GetService<IWeapon>();

}

public ActionResult Index()

{

return View(weapon);

}

}

Результат тот же.

### Итог:

Использование DI-контейнеров в современных приложениях необходимо, чтобы избавиться от сильной связности кода, и чтобы легко получать доступ из любой части кода к сервису. Также это необходимо для легкости написания Unit-тестов.