EBook Gratis

APRENDIZAJE castle-windsor

Free unaffiliated eBook created from
Stack Overflow contributors.

#castle-windsor

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: castle-windsor

It is an unofficial and free castle-windsor ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official castle-windsor.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con el castillo-windsor

Observaciones

Castle Windsor es un contenedor maduro de Inversión de Control disponible para .NET y Silverlight.

- La versión de lanzamiento actual es 3.3.0, lanzada en mayo de 2014. Consulte los enlaces a la derecha para descargarla desde GitHub o NuGet.
- La versión 4.0.0 beta de Castle Core se lanzó en julio de 2016.

A la web oficial del castillo.

A las páginas de documentación de Castle

Examples

Instalación

Castle Windsor está disponible a través de NuGet

1. Utilice los "Administrar paquetes de NuGet" y busque "Castle Windsor"

    - Para descargar para Visual Studio 2015
    - Para descargar versiones anteriores.

2. Use la consola del administrador de paquetes para ejecutar:

```
Install-Package Castle.Windsor
```

Ahora puedes usarlo para manejar dependencias en tu proyecto.

```
var container = new WindsorContainer(); // create instance of the container
container.Register(Component.For<IService>().ImplementedBy<Service>()); // register depndency
var service = container.Resolve<IService>(); // resolve with Resolve method
```

Consulte la documentación oficial para más detalles.

`Castle.Windsor` paquete `Castle.Windsor` depende del paquete `Castle.Core` y lo instalará también

Hola Mundo - Castillo Windsor

```
class Program
{
    static void Main(string[] args)
```

```
    {
        //Initialize a new container
        WindsorContainer container = new WindsorContainer();

        //Register IService with a specific implementation and supply its dependencies
        container.Register(Component.For<IService>()
                                    .ImplementedBy<SomeService>()
                                    .DependsOn(Dependency.OnValue("dependency", "I am Castle
Windsor")));

        //Request the IService from the container
        var service = container.Resolve<IService>();

        //Will print to console: "Hello World! I am Castle Windsor
        service.Foo();
}
```

Servicios:

```
public interface IService
{
    void Foo();
}

public class SomeService : IService
{
    public SomeService(string dependency)
    {
        _dependency = dependency;
    }

    public void Foo()
    {
        Console.WriteLine($"Hello World! {_dependency}");
    }

    private string _dependency;
}
```

Lea Empezando con el castillo-windsor en línea: https://riptutorial.com/es/castle-windsor/topic/5847/empezando-con-el-castillo-windsor

Capítulo 2: Estilos de vida

Examples

Estilos de vida estándar

Cuando un `Component` se resuelve desde el contenedor de Windsor, debe tener una definición del alcance en el que se encuentra. Por significado del alcance si y cómo se reutiliza y cuándo liberar el objeto para que lo destruya el recolector de basura. Este es el `LifeStlye` del `Component` .

La forma de especificar un estilo de vida es mediante el registro de un componente. Los dos `LifeStyles` más comunes son:

1. `Transient` : cada vez que el componente se resuelve, el contenedor produce una nueva instancia.

   ```
   Container.Register(Component.For<Bar>().LifestyleTransient());
   ```

2. `Singleton` : cada vez que se resuelve el componente, el contenedor devolverá la misma instancia.

   ```
   Container.Register(Component.For<Foo>().LifestyleSingleton());
   ```

   Singleton es el estilo de vida predeterminado, que se usará si no especifica ninguno explícitamente.

Otros `PerWebRequest LifeStyles` incorporados incluyen `PerWebRequest` , `Scoped` , `Bound` , `PerThread` , `Pooled`

Para obtener más detalles sobre los diferentes estilos de vida y para qué sirve cada uno, consulte la Documentación de Castle

Estilo de vida personalizado - IScopeAccessor

Al implementar su `IScopeAccessor` personalizado, puede crear diferentes tipos de ámbitos. Para el siguiente ejemplo, tengo las dos clases `Foo` y `Bar` en las cuales `Bar` se registrará con un estilo de `LifeStyle` personalizado.

Cada uno tiene una identificación para ayudar con las pruebas

```
public class Foo
{
    public Guid FooId { get; } = Guid.NewGuid();
}

public class Bar
{
    public Guid BarId { get; } = Guid.NewGuid();
```

```
}
```

Para registrar `Bar` como un `LifestyleScoped<T>` implementé `FooScopeAccessor` :

```
public class FooScopeAccessor : IScopeAccessor
{
    private static readonly ConcurrentDictionary<Foo, ILifetimeScope> collection = new
ConcurrentDictionary<Foo, ILifetimeScope>();

    public ILifetimeScope GetScope(CreationContext context)
    {
        return collection.GetOrAdd(context.AdditionalArguments["scope"] as Foo, new
DefaultLifetimeScope());
    }

    public void Dispose()
    {
        foreach (var scope in collection)
        {
            scope.Value.Dispose();
        }
        collection.Clear();
    }
}
```

Registro y Resolución:

```
WindsorContainer container = new WindsorContainer();
container.Register(Component.For<Foo>().LifestyleTransient());

var foo1 = container.Resolve<Foo>(); // FooId = 004350ac-40ff-4d1a-8022-7977f94eb418
var foo2 = container.Resolve<Foo>(); // FooId = 714aad8a-e4a2-4950-9017-e387c1c56133

container.Register(Component.For<Bar>().LifestyleScoped<FooScopeAccessor>());

var bar1 = container.Resolve<Bar>(new Dictionary<string, Foo> { ["scope"] = foo1 });
//                                              c144ba90-ce37-45c2-89d4-593d127fb723

var bar2 = container.Resolve<Bar>(new Dictionary<string, Foo> { ["scope"] = foo1 });
//                                              c144ba90-ce37-45c2-89d4-593d127fb723

var bar3 = container.Resolve<Bar>(new Dictionary<string, Foo> { ["scope"] = foo2 });
//                                              bcfe7ba4-cfb3-4b6e-8ecc-a3a3e5055bea

var bar4 = container.Resolve<Bar>(new Dictionary<string, Foo> { ["scope"] = foo1 });
//                                              c144ba90-ce37-45c2-89d4-593d127fb723
```

Como se ve arriba, `bar1` , `bar2` y `bar3` que se resolvieron con `Foo1` son todas referencias al mismo objeto, mientras que `bar4` se `bar4` con una nueva instancia de `Bar`

Para obtener más detalles sobre la implementación de un `IScopeAccessor` personalizado, consulte la documentación de Castle

Lea Estilos de vida en línea: https://riptutorial.com/es/castle-windsor/topic/7657/estilos-de-vida

Capítulo 3: Fábrica abstracta

Examples

Fábrica de validador de direcciones: requiere un parámetro para seleccionar una implementación

Escenario: debe seleccionar una implementación de validación de dirección cuando se envía un pedido de venta, y el validador está determinado por el país al que se envía el pedido. La fábrica necesita inspeccionar algún valor pasado como argumento para seleccionar la implementación correcta.

Primero, escribe una interfaz para la fábrica:

```
public interface IAddressValidatorFactory
{
    IAddressValidator GetAddressValidator(Address address);
    void Release(IAddressValidator created);
}
```

La interfaz indica que la implementación de `IAddressValidator` será determinada por la `Address` pase como argumento a `GetAddressValidator` .

Cuando registramos múltiples implementaciones de la misma interfaz con Windsor, generalmente las nombramos. Así que la fábrica tendrá que devolver el nombre de una implementación. En este caso, digamos que tenemos varias implementaciones de `IAddressValidator` registradas en nuestro contenedor:

```
container.Register(
    Component.For<IAddressValidator,UnitedStatesAddressValidator>()
        .Named("AddressValidatorFor_USA"),
    Component.For<IAddressValidator, FinlandAddressValidator>()
        .Named("AddressValidatorFor_FIN"),
    Component.For<IAddressValidator, MalawiAddressValidator>()
        .Named("AddressValidatorFor_MWI"),
    Component.For<IAddressValidator, CommonCountryAddressValidator>()
        .Named("FallbackCountryAddressValidator")
        .IsDefault()
    );
```

El trabajo de nuestra fábrica consistirá en tomar una `Address` y devolver el nombre de la implementación que Windsor resolverá. Eso requiere un *selector de componentes.*

```
public class AddressValidatorSelector : DefaultTypedFactoryComponentSelector
{
    public AddressValidatorSelector()
        : base(fallbackToResolveByTypeIfNameNotFound: true) { }

    protected override string GetComponentName(MethodInfo method, object[] arguments)
    {
```

```
        return "AddressValidatorFor_" + ((Address)arguments[0]).CountryCode;
    }
}
```

Esta clase le dice a Windsor que busque una implementación de `IAddressValidator` nombrada de acuerdo con el código de país del pedido. Para Finlandia eso es "AddressValidatorFor_Fin".
Y si no hay un validador para ese país específico, podemos recurrir a un validador predeterminado. También podríamos usar el mismo validador para varios países registrando la misma implementación más de una vez con nombres diferentes.

Escribimos el código para el selector de componentes, pero no escribimos el código para la propia fábrica. Windsor proporciona la fábrica, y le decimos que use nuestro `AddressValidatorSelector` .

```
// Add the factory facility once.
container.AddFacility<TypedFactoryFacility>();
container.Register(
    Component.For<IAddressValidatorFactory>()
        .AsFactory(new AddressValidatorSelector()));
```

Cuando una clase tiene una dependencia en `IAddressValidatorFactory` Windsor inyectará su propia implementación utilizando el selector de componentes que suministramos.

Fábrica abstracta simple - fábrica no toma parámetros

Escenario: debe resolver una dependencia cuando se llama a un método, *no* en el constructor.

Solución: inyectar una fábrica abstracta en el constructor. Cuando se llama al método, solicita la dependencia de la fábrica abstracta, que a su vez la resuelve desde el contenedor. (Su clase depende de la fábrica, pero nunca llama al contenedor en sí.)

Declara una interfaz para tu fábrica *:

```
public interface IFooFactory
{
    IFoo CreateFoo();
    void Release(IFoo foo);
}
```

Agregue el `TypedFactoryFacility` a su contenedor:

```
container.AddFacility<TypedFactoryFacility>();
```

Indique al contenedor que use `TypedFactoryFacility` para resolver dependencias en `IFooFactory` :

```
container.Register(
    Component.For<IFooFactory>().AsFactory(),
    Component.For<IFoo, MyFooImplementation>());
```

No es necesario crear una instancia de una fábrica. Windsor hace eso.

Ahora puedes inyectar la fábrica en una clase y usarla así:

```
public class NeedsFooFactory
{
    private readonly IFooFactory _fooFactory;

    public NeedsFooFactory(IFooFactory fooFactory)
    {
        _fooFactory = fooFactory;
    }

    public void MethodThatNeedsFoo()
    {
        var foo = _fooFactory.CreateFoo();
        foo.DoWhatAFooDoes();
        _fooFactory.Release(foo);
    }
}
```

Al llamar al método `Release` , el contenedor libera el componente que resolvió. De lo contrario, no se publicará hasta que se `_fooFactory` (que es cuando se lanza `NeedsFooFactory`).

* Windsor deduce qué método es el método "crear" y cuál es el método "liberar". Si un método devuelve algo, se asume que el contenedor debe resolverlo. Si un método no devuelve nada (`void`), entonces es el método de "lanzamiento".

Lea Fábrica abstracta en línea: https://riptutorial.com/es/castle-windsor/topic/6727/fabrica-abstracta

Capítulo 4: Instaladores

Examples

Conceptos básicos - Crear y usar un instalador

Los instaladores son tipos personalizados que implementan la interfaz `IWindsorInstaller` y se utilizan para registrar los `Component` en el contenedor, utilizando la API de registro fluida.

```
public class MyInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(Component.For<IMyType>()
                                    .ImplementedBy<ConcreteMyType1>());

        //Registering several components in one call to .Register
        container.Register(
            Component.For<IFoo>().ImplementedBy<Foo>(),
            Component.For<IBar>().ImplementedBy<Bar());
    }
}

//To use the installer:
WindsorContainer container = new WindsorContainer();
container.Install(new MyInstaller());

container.Resolve<IFoo>();
```

Tenga en cuenta

> Los instaladores deben tener un constructor público predeterminado: cuando los instaladores son instanciados por Windsor, deben tener un constructor público predeterminado. De lo contrario se lanzará una excepción.

Clase de montaje

Otra forma de `.Install` instaladores es usando la clase `FromAssembly` de Castle. Proporciona una serie de funciones para ubicar a los instaladores en los ensamblados cargados. Por ejemplo:

```
//Will locate IInstallers in the current assembly that is calling the method
container.Install(FromAssembly.This());
```

Para más detalles ver la documentación del castillo.

Instalación desde la configuración

Castle permite registrar componentes también a través del registro XML.

```
//To install from the app/web.config
```

```
container.Install(Configuration.FromAppConfig());

//To install from an xml file
Configuration.FromXmlFile("relative_path_to_file.xml");
```

Lea la documentación de Castle para "¿ Para qué sirve?"

Lea Instaladores en línea: https://riptutorial.com/es/castle-windsor/topic/6243/instaladores

Capítulo 5: Interceptores

Examples

Creando interceptores personalizados.

Los interceptores requieren la interfaz `IInterceptor` . Cualquier método público dentro de la clase interceptada será interceptado *(incluyendo captadores y definidores)*

```
public class MyInterceptor : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
       //Calls the next method in the chain - The last one will be the
       //original method that was intercepted
       invocation.Proceed();
    }
}
```

Lea esto para obtener información sobre el registro de un interceptor en componentes

Los interceptores no tienen que llamar Proceder cada vez

Los interceptores son una buena herramienta para implementar inquietudes transversales , como el registro o la autenticación. Digamos que tenemos un siguiente servicio:

```
public interface IService
{
    string CreateOrder(NetworkCredential credentials, Order orderToCreate);
    string DeleteOrder(NetworkCredential credentials, int orderId);
}

public class Service : IService
{
    public string CreateOrder(NetworkCredential credentials, Order orderToCreate)
    {
        // ...

        return "Order was created succesfully.";
    }

    public string DeleteOrder(NetworkCredential credentials, int orderId)
    {
        // ...

        return "Order was deleted succesfully.";
    }
}
```

Podemos crear el siguiente interceptor:

```
public class AuthorizationInterceptor : IInterceptor
```

```
{
    public void Intercept(IInvocation invocation)
    {
        var userCredentials = invocation.Arguments[0] as NetworkCredential;

        if (userCredentials.UserName == "tom" && userCredentials.Password == "pass123")
            // this ^ verification is obviously silly, never do real security like this
        {
            invocation.Proceed();
        }
        else
        {
            invocation.ReturnValue = $"User '{userCredentials.UserName}' was not
authenticated.";
        }
    }
}
```

Que se puede registrar y utilizar así:

```
var container = new WindsorContainer();
container.Register(
    Component.For<AuthorizationInterceptor>(),

Component.For<IService>().ImplementedBy<Service>().Interceptors<AuthorizationInterceptor>());

var service = container.Resolve<IService>();

System.Diagnostics.Debug.Assert(
    service.DeleteOrder(new NetworkCredential { UserName = "paul", Password = "pass321" }, 8)
        == "User 'paul' was not authenticated.");

System.Diagnostics.Debug.Assert(
    service.CreateOrder(new NetworkCredential { UserName = "tom", Password = "pass123" }, new
Order())
        == "Order was created succesfully.");
```

La lección importante es que el interceptor puede decidir pasar la llamada y, si no lo hace, puede proporcionar un valor de retorno arbitrario utilizando la propiedad `ReturnValue` .

Los interceptores pueden ser encadenados en un orden fijo

Para registrarse como este:

```
var container = new WindsorContainer();
container.Register(
    Component.For<FirstInterceptor>(),
    Component.For<SecondInterceptor>(),
    Component.For<ThirdInterceptor>(),

    Component.For<IService>()
        .ImplementedBy<Service>()
        .Interceptors<FirstInterceptor>()
        .Interceptors<SecondInterceptor>()
        .Interceptors<ThirdInterceptor>());

var service = container.Resolve<IService>();
```

```
service.CreateOrder(new Order());
```

Con interceptores en el siguiente espíritu:

```
public class FirstInterceptor : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("First!");
        invocation.Proceed();
        Console.WriteLine("First.");
    }
}
```

Y con el servicio implementado de la siguiente manera:

```
public interface IService
{
    void CreateOrder(Order orderToCreate);
}

public class Service : IService
{
    public void CreateOrder(Order orderToCreate)
    {
        Console.WriteLine("Creating order...");

        // ...
    }
}
```

Podemos observar una salida que demuestra que los interceptores creados en un orden fijo se ejecutarán en ese orden:

```
First!
Second!
Third!
Creating order...
Third.
Second.
First.
```

Los interceptores pueden tener dependencias.

Los interceptores están registrados como componentes regulares en Windsor. Al igual que otros componentes, pueden depender de otros componentes.

Con el siguiente servicio para validar credenciales:

```
public interface ICredentialsVerifier
{
    bool IsAuthorizedForService(NetworkCredential credentials);
}
```

```
public class MockCredentialsVerifier : ICredentialsVerifier
{
    public bool IsAuthorizedForService(NetworkCredential credentials)
        => credentials.UserName == "tom" && credentials.Password == "pass123";
        // this ^ verification is obviously silly, never do real security like this
}
```

Podemos utilizar el siguiente interceptor:

```
public class AuthorizationInterceptor : IInterceptor
{
    private readonly ICredentialsVerifier _credentialsVerifier;

    public AuthorizationInterceptor(ICredentialsVerifier credentialsVerifier)
    {
        _credentialsVerifier = credentialsVerifier;
    }

    public void Intercept(IInvocation invocation)
    {
        var userCredentials = invocation.Arguments[0] as NetworkCredential;

        if (_credentialsVerifier.IsAuthorizedForService(userCredentials))
        {
            invocation.Proceed();
        }
        else
        {
            invocation.ReturnValue = $"User '{userCredentials.UserName}' was not
authenticated.";
        }
    }
}
```

Solo tenemos que registrarlo correctamente en la raíz de la composición de esta manera:

```
var container = new WindsorContainer();
container.Register(
    Component.For<AuthorizationInterceptor>(),

    Component.For<ICredentialsVerifier>().ImplementedBy<MockCredentialsVerifier>(),

Component.For<IService>().ImplementedBy<Service>().Interceptors<AuthorizationInterceptor>());

var service = container.Resolve<IService>();
```

Lea Interceptores en línea: https://riptutorial.com/es/castle-windsor/topic/6663/interceptores

Capítulo 6: Registro API fluido

Examples

Registro de varios tipos de la misma interfaz

Al registrar tipos en el contenedor, `Castle` utiliza el tipo de la clase para resolver. En el caso de que haya más de un registro para un tipo específico, la propiedad `Name` debe establecerse:

```
public void Install(IWindsorContainer container, IConfigurationStore store)
{
    container.Register(
        Component.For<IFoo>().ImplementedBy<Foo>().Named("Registration1"),
        Component.For<IBar>().ImplementedBy<Bar().Named("Registration2"));
}
```

Especificando el estilo de vida

Un `Lifestyle` es el "cómo" Castle controla el alcance en el que se utiliza un componente y cuándo se debe limpiar. Los estilos de vida `Singelton` son `Singelton` , `Transient` , `PerWebRequest` , `Scoped` , `Bound` , `PerThread` y `Pooled`

```
container.Register(
   Component.For<IFoo>()
            .ImplementedBy<Foo>()
            .LifestyleSingleton(),

    Component.For<IBar>()
             .ImplementedBy<Bar>()
             .LifestyleTransient());
```

DependsOn - Especifique Dependencias

La idea completa con la inyección de dependencia es que una clase no crea instancias de sus dependencias, sino que las solicita (a través del constructor o la propiedad). Usar Castle para especificar la manera de resolver una dependencia es usar `DependsOn` :

```
public class Foo : IFoo
{
    public Foo(IBar bar, string val)
    {
        Bar = bar;
        Val = val;
    }
    public IBar Bar { get; set; }
    public string Val { get; set; }
}

container.Register(
    Component.For<IBar>().ImplementedBy<Bar>().Named("bar1"),
```

```
    Component.For<IBar>().ImplementedBy<Bar>().Named("bar2"),

    Component.For<IFoo>()
             .ImplementedBy<Foo>()
             .DependsOn(Dependency.OnComponent("bar", "bar1"),
                        Dependency.OnValue("val","some value")));
```

Interceptores

Al registrar un componente, use el método `Interceptors()` para especificar cuáles son los interceptores / tipos de interceptores que se utilizarán para este componente:

El `TInterceptor` debe implementar la interfaz `IInterceptor`

Un único interceptor por tipo:

```
container.Register(
    Component.For<MyInterceptor>(),
    Component.For<IFoo>()
             .ImplementedBy<Foo>()
             .Interceptors<MyInterceptor>());
```

Dos interceptores por tipo:

```
container.Register(
    Component.For<MyInterceptor1>(),
    Component.For<MyInterceptor2>(),
    Component.For<IFoo>()
             .ImplementedBy<Foo>()
             .Interceptors<MyInterceptor1, MyInterceptor2>());
```

Más de 2 interceptores por tipo:

```
container.Register(
    Component.For<MyInterceptor1>(),
    Component.For<MyInterceptor2>(),
    Component.For<MyInterceptor3>(),
    Component.For<IFoo>()
             .ImplementedBy<Foo>()
             .Interceptors(typeof(MyInterceptor1),
                           typeof(MyInterceptor2),
                           typeof(MyInterceptor3)));
```

Lea Registro API fluido en línea: https://riptutorial.com/es/castle-windsor/topic/7235/registro-api-fluido

Creditos

S. No	Capítulos	Contributors
1	Empezando con el castillo-windsor	Community, Edu, Gilad Green, Liam, NikolayKondratyev
2	Estilos de vida	Gilad Green
3	Fábrica abstracta	RamenChef, Scott Hannen
4	Instaladores	Gilad Green
5	Interceptores	4444, Gilad Green, Lukáš Lánský
6	Registro API fluido	Gilad Green