FREE eBook

LEARNING castle-windsor

Free unaffiliated eBook created from
Stack Overflow contributors.

#castle-windsor

Table of Contents

About

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

Chapter 1: Getting started with castle-windsor

Remarks

Castle Windsor is a mature Inversion of Control container available for .NET and Silverlight.

- The current release version is 3.3.0, released in May 2014. Refer to the links on the right to download it from GitHub or NuGet.
- Castle Core version 4.0.0 beta was released in July 2016.

To Castle's official website

To Castle's Documentation Pages

Examples

Installation

Castle Windsor is available via NuGet

1. Use the "Manage NuGet Packages" and search for "castle windsor"

    - To download for Visual Studio 2015
    - To download for previous versions

2. Use Package Manager Console to execute:

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

Now you can use it to handle dependencies in your project.

```
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
```

See official documentation for more details.

`Castle.Windsor` package depends on `Castle.Core` package and it will install it too

Hello World - Castle 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();
}
```

Services:

```
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;
}
```

Read Getting started with castle-windsor online: https://riptutorial.com/castle-windsor/topic/5847/getting-started-with-castle-windsor

Chapter 2: Abstract Factory

Examples

Address validator factory - requires a parameter to select an implementation

Scenario: You need to select an implementation of address validation when a sales order is submitted, and the validator is determined by the country to which the order is shipping. The factory needs to inspect some value passed as an argument to select the correct implementation.

First, write an interface for the factory:

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

The interface indicates that the implementation of `IAddressValidator` will be determined by the `Address` passed as an argument to `GetAddressValidator`.

When we register multiple implementations of the same interface with Windsor we typically name them. So the factory will need to return the name of an implementation. In this case, let's say we have several implementations of `IAddressValidator` registered with our container:

```
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()
    );
```

The job of our factory will to take an `Address` and return the name of the implementation that Windsor will resolve. That requires a *component selector.*

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

    protected override string GetComponentName(MethodInfo method, object[] arguments)
    {
        return "AddressValidatorFor_" + ((Address)arguments[0]).CountryCode;
    }
}
```

This class tells Windsor to look for an implementation of `IAddressValidator` named according to the country code of the order. For Finland that's "AddressValidatorFor_Fin".
And if there is no validator for that specific country we can fall back to a default validator. We could also use the same validator for multiple countries by registering the same implementation more than once with different names.

We wrote the code for the component selector, but we don't write the code for the factory itself. Windsor provides the factory, and we tell it to use our `AddressValidatorSelector`.

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

When a class has a dependency on `IAddressValidatorFactory` Windsor will inject its own implementation using the component selector we supplied.

Simple abstract factory - factory takes no parameters

Scenario: You need to resolve a dependency when a method is called, *not* in the constructor.

Solution: Inject an abstract factory into the constructor. When the method is called, it requests the dependency from the abstract factory, which in turn resolves it from the container. (Your class depends on the factory but never calls the container itself.)

Declare an interface for your factory*:

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

Add the `TypedFactoryFacility` to your container:

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

Instruct the container to use the `TypedFactoryFacility` to resolve dependencies on `IFooFactory`:

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

You don't need to create an instance of a factory. Windsor does that.

You can now inject the factory into a class and use it like this:

```
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);
    }
}
```

Calling the `Release` method causes the container to release the component it resolved. Otherwise it won't be released until `_fooFactory` is released (which is whenever `NeedsFooFactory` is released.)

*Windsor infers which method is the "create" method and which is the "release" method. If a method returns something then it's assumed that the container must resolve it. If a method returns nothing (`void`) then it's the "release" method.

Read Abstract Factory online: https://riptutorial.com/castle-windsor/topic/6727/abstract-factory

Chapter 3: Fluent API Registration

Examples

Registering Several Types of Same Interface

When registering types to the container `Castle` uses the type of the class in order to resolve. In the case that there is more than one registration for a specific type the `Name` property must be set:

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

Specifying Lifestyle

A `Lifestyle` is the "how" Castle controls the scope in which a component is used and when to clean it up. The built-in lifestyles are `Singelton`, `Transient`, `PerWebRequest`, `Scoped`, `Bound`, `PerThread` and `Pooled`

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

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

DependsOn - Specify Dependencies

The entire idea with dependency injection is that a class does not instantiate its dependencies but requests them (through constructor or property). Using Castle the way for specifying the way to resolve a dependency is by using the `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")));
```

Interceptors

When registering a component use the `Interceptors()` method to specify what are the interceptors/types of interceptors to be used for this component:

The `TInterceptor` must implement the `IInterceptor` interface

A single interceptor by type:

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

Two interceptors by type:

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

More than 2 interceptors by type:

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

Read Fluent API Registration online: https://riptutorial.com/castle-windsor/topic/7235/fluent-api-registration

Chapter 4: Installers

Examples

Basics - Creating and using an Installer

Installers are custom types that implement the `IWindsorInstaller` interface and are used to register `Component`s to the container, using the fluent registration API.

```
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>();
```

Keep in mind

> Installers must have public default constructor: When installers are instantiated by Windsor, they must have public default constructor. Otherwise an exception will be thrown.

FromAssembly Class

Another way to `.Install` installers is by using Castle's `FromAssembly` class. It gives an array of functions to locate installers in the loaded assemblies. For example:

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

For more details see Castle's Documentation

Installing from Configuration

Castle enables to register components also via XML Registration.

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

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

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

Read Castle's documentation for "What is it for"

Read Installers online: https://riptutorial.com/castle-windsor/topic/6243/installers

Chapter 5: Interceptors

Examples

Creating custom interceptors

Interceptors require the `IInterceptor` interface. Any public method within the intercepted class will be intercepted *(including getters and setters)*

```
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();
    }
}
```

Read this for information about registering an interceptor to components

Interceptors doesn't have to call Proceed every time

Interceptors are a good tool for implementing cross-cutting concerns such as logging or authentication. Let's say we have a following service:

```
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.";
    }
}
```

We can create following 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.";
        }
    }
}
```

That can be registered and utilized like this:

```
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.");
```

The important lesson is that interceptor can decide to pass call and if it doesn't, it can supply arbitrary return value using `ReturnValue` property.

Interceptors can be chained in a fixed order

For registration like this:

```
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());
```

With interceptors in the following spirit:

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

And with service implemented in the following way:

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

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

        // ...
    }
}
```

We can observe output that demonstrates interceptors created in a fixed order are going to be executed in that order:

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

Interceptors can have dependencies

Interceptors are registered like regular components in Windsor. Like other components, they can depend on another components.

With following service for validating credentials:

```
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
}
```

We can use the following 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.";
        }
    }
}
```

We just have to properly register it in the composition root like this:

```
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>();
```

Read Interceptors online: https://riptutorial.com/castle-windsor/topic/6663/interceptors

Chapter 6: Lifestyles

Examples

Standard Lifestyles

When a `Component` is resolved from the Windsor container it must have a definition of the scope it is in. By scope meaning if and how it is reused and when to release the object for the Garbage Collector to destroy. This is the `LifeStlye` of the `Component`.

The way to specify a LifeStyle is by registering a component. The two most common `LifeStyles` are:

1. `Transient` - Each time the component is resolved a new instance of it is produced by the container.

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

2. `Singleton` - Each time the component is resolved the same instance will be returned by the container

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

   Singleton is the default lifestyle, which will be use if you don't specify any explicitly.

Other built-in `LifeStyles` include `PerWebRequest`,`Scoped`, `Bound`, `PerThread`, `Pooled`

For more details about the different lifestyles and for what each is good for, refer to Castle's Documentation

Custom LifeStyle - IScopeAccessor

By implementing your custom `IScopeAccessor` you can create different types of scopes. For the following example I have the two classes `Foo` and `Bar` in which `Bar` will be registered with a custom `LifeStyle`.

Each have an Id to assist with testing

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

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

To register `Bar` as a `LifestyleScoped<T>` I implemented `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();
    }
}
```

Registering and Resolving:

```
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
```

As seen above `bar1`, `bar2` and `bar3` which were Resolved using `Foo1` are all reference to the same object while `bar4` has been Resolved with a new instance of `Bar`

For more details about implementing a custom `IScopeAccessor` refer to Castle's Documentation

Read Lifestyles online: https://riptutorial.com/castle-windsor/topic/7657/lifestyles

Credits

S. No	Chapters	Contributors
1	Getting started with castle-windsor	Community, Edu, Gilad Green, Liam, NikolayKondratyev
2	Abstract Factory	RamenChef, Scott Hannen
3	Fluent API Registration	Gilad Green
4	Installers	Gilad Green
5	Interceptors	4444, Gilad Green, Lukáš Lánský
6	Lifestyles	Gilad Green