‹ Previous Next ›

Unity Container: Register and Resolve

In the previous section, we installed Unity framework in our console project. Here, we will learn how to register type-mapping and resolve it using the unity container.

As we learned in the IoC container chapter, every container must provide a way to register and resolve dependencies. Unity container provides the `RegisterType()` and `Resolve()` methods for this.

We are going to use the following sample classes to demo the registration and resolution of dependencies throughout this chapter.

Example: C# Copy

```
public interface ICar
{
    int Run();
}

public class BMW : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}

public class Ford : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }
}

public class Audi : ICar
{
    private int _miles = 0;

    public int Run()
    {
        return ++_miles;
    }

}
public class Driver
{
    private ICar _car = null;

    public Driver(ICar car)
    {
        _car = car;
    }
```

```
    public void RunCar()
    {
        Console.WriteLine("Running {0} - {1} mile ", _car.GetType().Name, _car.Run());
    }
}
```

As you can see in the sample classes, the `Driver` class depends on the `ICar` interface. So, when we instantiate the `Driver` class object, then we will have to pass an instance of a class which implements the `ICar` interface, such as the `BMW`, `Audi` or `Ford` class as shown below.

Example: C# Copy

```
Driver driver = new Driver(new BMW());

driver.RunCar();
```

Output:

```
Running BMW - 1 mile
```

In the above example, we created and passed an object of `BMW` while creating an object of the `Driver` class. Thus, we injected the dependency of the `Driver` class manually. Now, we will use Unity container to understand different ways to register and resolve dependencies.

Using UnityContainer

In order to use Unity container, we first need to create an object of it. You can use any class which implements the `IUnityContainer` interface. Unity container includes the `UnityContainer` class in the `Microsoft.Practices.Unity` namespace that implements the `IUnityContainer` interface. If you need to extend the container, then you can create your own custom class and implement the `IUnityContainer` interface as per your need.

Example: Instantiate UnityContainer - C# Copy

```
using Microsoft.Practices.Unity;

IUnityContainer container = new UnityContainer();
//or
var container = new UnityContainer();
```

Next, we need to register type-mapping.

Register

Before Unity resolves the dependencies, we need to register the type-mapping with the container, so that it can create the correct object for the given type. Use the `RegisterType()` method to register a type mapping. Basically, it configures which class to instantiate for which interface or base class. For example, if we want Unity container to create and supply an object of the `BMW` class whenever it needs to supply a dependency of the `ICar` interface, then we first need to register it as shown below.

Example: Register Type with Unity - C# Copy

```
IUnityContainer container = new UnityContainer();

container.RegisterType<ICar, BMW>();
```

Here, `container.RegisterType<ICar, BMW>()` requests Unity to create an object of the `BMW` class and inject it through a constructor whenever you need to inject an object of `ICar`.

The RegisterType method includes many overloads. Learn about all the overloads of RegisterType on MSDN.

So now, after registration, we can use the `Resolve()` method.

Resolve

Unity creates an object of the specified class and automatically injects the dependencies using the `resolve()` method. We have registered `BMW` with `ICar` above. Now, we can instantiate the `Driver` class using Unity container without using the `new` keyword as shown below.

Example: Resolve - C# Copy

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ICar, BMW>();// Map ICar with BMW

//Resolves dependencies and returns the Driver object
Driver drv = container.Resolve<Driver>();
drv.RunCar();
```

Output:

```
Running BMW - 1 mile
```

In the above example, Unity container creates an object of the `Driver` class using the `container.Resolve<driver>()` method. The `Driver` class is a dependency of `ICar`. So, `container.Resolve<Driver>()` returns an object of the `Driver` class by automatically creating and injecting a `BMW` object in it. All this is behind the scene. The `BMW` object is created and injected because we register the `BMW` type with `ICar`.

Unity container will create a new object and inject it every time we resolve the same type.

```
var container = new UnityContainer();
container.RegisterType<ICar, BMW>();

Driver driver1 = container.Resolve<Driver>();
driver1.RunCar();

Driver driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:

```
Running BMW - 1 mile
Running BMW - 1 mile
```

In the above example, container injects the `BMW` object whenever it resolves the `Driver` class, e.g. driver1 and driver2 both have references to separate `BMW` objects.

Thus, you can create an object of the specified type using Unity container. Learn about all the overloads of the Resolve method on MSDN.

Multiple Registration

Unity container will inject the last registered type if you register multiple mappings of the same type.

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ICar, BMW>();
container.RegisterType<ICar, Audi>();

Driver driver = container.Resolve<Driver>();
driver.RunCar();
```

Output:

```
Running Audi - 1 Mile
```

In the above example, `ICar` is mapped to both `BMW` and `Audi`. However, Unity will inject `Audi` every time because it has been registered last.

Register Named Type

You can register a type-mapping with a name which you can use with the `Resolve()` method.

Example: Register Named Type - C# Copy

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ICar, BMW>();
container.RegisterType<ICar, Audi>("LuxuryCar");

ICar bmw = container.Resolve<ICar>();  // returns the BMW object
ICar audi = container.Resolve<ICar>("LuxuryCar"); // returns the Audi object
```

As you can see above, we have mapped `ICar` with both the `BMW` and the `Audi` class. However, we have given the name "LuxuryCar" to the ICar-Audi mapping. So now, the `Resolve()` method will return an object of Audi if we specify the mapping name.

Consider the following example:

```
var container = new UnityContainer();
container.RegisterType<ICar, BMW>();
container.RegisterType<ICar, Audi>("LuxuryCar");

// Registers Driver type
container.RegisterType<Driver>("LuxuryCarDriver",
                new InjectionConstructor(container.Resolve<ICar>("LuxuryCar")));

Driver driver1 = container.Resolve<Driver>();// injects BMW
driver1.RunCar();

Driver driver2 = container.Resolve<Driver>("LuxuryCarDriver");// injects Audi
driver2.RunCar();
```

Output:

```
Running BMW - 1 Mile
Running Audi - 1 Mile
```

In the above example, we registered the `Driver` class with the name "LuxuryCarDriver" and specified an object of `InjectionConstructor`. The `new InjectionConstructor(container.Resolve<ICar>("LuxuryCar"))` specifies a construction injection

for the `Driver` class, which passes an object of `Audi` because `container.Resolve("LuxuryCar")` returns an `Audi` object. So now, we can use `container.Resolve<Driver>("LuxuryCarDriver")` to resolve the `Driver` with `Audi` even if `ICar` is registered with `BMW` by default.

Register Instance

Unity container allows us to register an existing instance using the `RegisterInstance()` method. It will not create a new instance for the registered type and we will use the same instance every time.

```
var container = new UnityContainer();
ICar audi = new Audi();
container.RegisterInstance<ICar>(audi);

Driver driver1 = container.Resolve<Driver>();
driver1.RunCar();
driver1.RunCar();

Driver driver2 = container.Resolve<Driver>();
driver2.RunCar();
```

Output:

```
Running Audi - 1 Mile
Running Audi - 2 Mile
Running Audi - 3 Mile
```

Thus, we can register and resolve different types using Unity container. Learn how Unity container performs constructor injection in the next chapter.

‹ Previous Next ›