OPEN SOURCE

INTRODUCING UNITY.WCF - PROVIDING EASY IOC INTEGRATION FOR YOUR WCF SERVICES

There are numerous blog posts available that explain how to hook up Unity with WCF. Unfortunately, many of these are incomplete, too basic or just plain wrong. Additionally, as far as I can tell, nobody has created a NuGet package to get you up and running as quickly as possible. This post introduces Unity.WCF, an open source NuGet package that (hopefully) just works in most situations, deals with cleaning up IDisposable instances and also provides a nice mechanism for automatically adding WCF behaviors to your services.

INSTANTIATING A SERVICE CONTRACT WITHOUT A PARAMETERLESS CONSTRUCTOR

Out of the box, all service contracts require a default constructor which is obviously somewhat troublesome if you want to inject your dependencies. Making WCF work with IoC is not difficult but it does involve the creation of a few classes in order to hook into the WCF pipeline at the appropriate point. WCF is full of extensibility points and when it comes to service instantiation, the *IInstanceProvider* interface is what you need.

```
public interface IInstanceProvider
{
  object GetInstance(InstanceContext instanceContext);
  object GetInstance(InstanceContext instanceContext, Message message);
```

Do you need professional PDFs? Try PDFmyURL!

```
  void ReleaseInstance(InstanceContext instanceContext, object instance);
}
```

So, GetInstance needs to create the service instance and ReleaseInstance needs to tidy up afterwards. Virtually all implementations available on the web completely ignore the ReleaseInstance method or just call the TearDown method on the Unity container which looks as though it might do something useful, but in fact does absolutely nothing. The easiest way to dispose of items in Unity on a per client basis is to use child containers. This is exactly how our Unity.Mvc3 library works. Unity.WCF takes a similar approach. The complete UnityInstanceProvider is displayed below:

```
public class UnityInstanceProvider : IInstanceProvider
{
  private readonly IUnityContainer _container;
  private readonly Type _contractType;

  public UnityInstanceProvider(IUnityContainer container, Type contractType)
  {
    if (container == null)
    {
      throw new ArgumentNullException("container");
    }

    if (contractType == null)
    {
      throw new ArgumentNullException("contractType");
    }

    _container = container;
    _contractType = contractType;
  }

  public object GetInstance(InstanceContext instanceContext, Message message)
  {
    var childContainer =

instanceContext.Extensions.Find<UnityInstanceContextExtension>().GetChildContainer(_container)
;

    return childContainer.Resolve(_contractType);
  }

  public object GetInstance(InstanceContext instanceContext)
  {
    return GetInstance(instanceContext, null);
  }
```

Do you need professional PDFs? Try **PDFmyURL!**

```
  public void ReleaseInstance(InstanceContext instanceContext, object instance)
  {
instanceContext.Extensions.Find<UnityInstanceContextExtension>().DisposeOfChildContainer();
  }
}
```

The code is quite self explanatory. We create a new child container whenever a new connection is made and then use the child container to resolve the service implementation. When the connection is terminated, we dispose of the child container, forcing all IDisposable dependencies to also be disposed (or to be more accurate, all dependencies registered with a HierarchicalLifetimeManager are disposed).

Because a single UnityInstanceProvider is created for the service, we cannot store the child container in a private member variable. Instead, we make use of WCF extensions to store the container on a per client basis. Check out the code on **CodePlex** if you are interested in this.

LETTING WCF KNOW ABOUT OUR CUSTOM IINSTANCEPROVIDER

The next thing we need to do is to get WCF to actually use our instance provider. As is often the case with WCF, this is achieved through behaviors.

```
public class UnityContractBehavior : IContractBehavior
{
  private readonly IInstanceProvider _instanceProvider;

  public UnityContractBehavior(IInstanceProvider instanceProvider)
  {
    if (instanceProvider == null)
    {
      throw new ArgumentNullException("instanceProvider");
    }

    _instanceProvider = instanceProvider;
  }

  public void AddBindingParameters(ContractDescription contractDescription, ServiceEndpoint
endpoint, BindingParameterCollection bindingParameters)
  {
```

Do you need professional PDFs? Try **PDFmyURL!**

```
  }

  public void ApplyClientBehavior(ContractDescription contractDescription, ServiceEndpoint
endpoint, ClientRuntime clientRuntime)
  {
  }

  public void ApplyDispatchBehavior(ContractDescription contractDescription, ServiceEndpoint
endpoint, DispatchRuntime dispatchRuntime)
  {
    dispatchRuntime.InstanceProvider = _instanceProvider;
    dispatchRuntime.InstanceContextInitializers.Add(new UnityInstanceContextInitializer());
  }

  public void Validate(ContractDescription contractDescription, ServiceEndpoint endpoint)
  {
  }
}
```

As you can see, the behavior has very little in the way of implementation and most interface methods are empty. All we needs to do is set the InstanceProvider to our injected Implementation in the ApplyDispatchBehavior method. We also add a custom InstanceContextInitializer which is used to hook up the "child container holding" extension that we discussed above. This is not the end of the story however. How do we add the behavior to the service contract (or more accurately, to all the contracts within the service)?

SERVICEHOST / SERVICEHOSTFACTORY - THE ROOT OF A WCF APPLICATION

The answer is... that it depends. In order to add our behavior and configure our IoC container, we need access to the root of the WCF application - the compositional root. Depending upon how you are hosting your WCF service, this can be seen as either the ServiceHost or the ServiceHostFactory. We will subclass both of these to cover WAS and Windows Service hosted scenarios.

```
public abstract class UnityServiceHostFactory : ServiceHostFactory
{
  protected abstract void ConfigureContainer(IUnityContainer container);

  protected override ServiceHost CreateServiceHost(Type serviceType, Uri[] baseAddresses)
```

Do you need professional PDFs? Try **PDFmyURL!**

```
  {
    var container = new UnityContainer();

    ConfigureContainer(container);

    return new UnityServiceHost(container, serviceType, baseAddresses);
  }
}
```

The UnityServiceHostFactory is declared as an abstract class with an abstract ConfigureContainer method. This allows users of the library to subclass it in their WAS hosted WCF services, implement ConfigureContainer and add their Unity registration code. It effectively exposes an entry point into the service which is otherwise unavailable. Internally, we instantiate a new UnityServiceHost, passing in the configured container.

```
public class UnityServiceHost : ServiceHost
{
  public UnityServiceHost(IUnityContainer container, Type serviceType, params Uri[]
baseAddresses)
    : base(serviceType, baseAddresses)
  {
    if (container == null)
    {
      throw new ArgumentNullException("container");
    }

    ApplyServiceBehaviors(container);

    ApplyContractBehaviors(container);

    foreach (var contractDescription in ImplementedContracts.Values)
    {
      var contractBehavior =
        new UnityContractBehavior(new UnityInstanceProvider(container,
contractDescription.ContractType));

      contractDescription.Behaviors.Add(contractBehavior);
    }
  }

  private void ApplyContractBehaviors(IUnityContainer container)
  {
    var registeredContractBehaviors = container.ResolveAll<IContractBehavior>();

    foreach (var contractBehavior in registeredContractBehaviors)
```

Do you need professional PDFs? Try **PDFmyURL!**

```
    {
      foreach (var contractDescription in ImplementedContracts.Values)
      {
        contractDescription.Behaviors.Add(contractBehavior);
      }
    }
  }

  private void ApplyServiceBehaviors(IUnityContainer container)
  {
    var registeredServiceBehaviors = container.ResolveAll<IServiceBehavior>();

    foreach (var serviceBehavior in registeredServiceBehaviors)
    {
      Description.Behaviors.Add(serviceBehavior);
    }
  }
}
```

The UnityServiceHost loops around all contracts within the service and adds our custom behavior to each. The ApplyServiceBehaviors and ApplyContractBehaviors add the ability to register behaviors with Unity and have them automatically be applied by the ServiceHost.

AUTOMATIC BEHAVIORS

Any implementations of IServiceBehavior or IContractBehavior that are registered with Unity will automatically be applied by the custom ServiceHost. Why is this useful? You can already apply behaviors though configuration and attributes but these methods do not allow you to inject dependencies into your behaviors. With Unity.WCF, this is trivial.

```
container
  .RegisterType<IContractBehavior, ErrorBehavior>("errorHandlerBehavior")
  .RegisterType<IErrorLogger, AnErrorLogger>();
```

In the above example, the ErrorBehavior class has a constructor that takes in an IErrorLogger. Since an implementation of this interface is also registered with Unity, the ServiceHost will be able to instantiate the ErrorBehavior and add it to the service automatically.

The only thing to be aware of is that these behavior registration must be named. This is because internally, we use

Do you need professional PDFs? Try **PDFmyURL!**

container.ResolveAll<...> which only returns named instances.

```
container
  .RegisterType<IServiceBehavior, ValidateDataAnnotationsBehavior>("validationBehavior")
  .RegisterType<IServiceBehavior, SomeOtherBehavior>();
```

In the example registration displayed above, only the ValidateDataAnnotationsBehavior will be added. SomeOtherBehavior is not named so will be ignored. Note that the registration name itself is not important. Just having ANY name is sufficient.

ADDING UNITY.WCF TO YOUR PROJECT

Using the Package Manager Console, type **install-package Unity.WCF**. Alternatively, you can use the Package Manager GUI and search for Unity.WCF. The Unity.WCF NuGet package will automatically add all the necessary references including the Unity NuGet package. Once installed, your next action depends on how you are hosting your service.

WAS / IIS HOSTED SERVICES

For IIS/WAS-based hosting, right click on your svc file in the solution explorer and select View Markup. Next replace CodeBehind="Service1.svc.cs" with Factory="WcfService1.WcfServiceFactory", where WcfService1 is the namespace of your project. If you are using fileless activation and do not have an SVC file, change your web.config instead.

```
<serviceHostingEnvironment>
  <serviceActivations>
    <add factory="WcfService1.WcfServiceFactory" relativeAddress="./Service1.svc"
service="WcfService1.Service1"/>
  </serviceActivations>
</serviceHostingEnvironment>
```

Open the WcfServiceFactory class that has been added to the root of your application. Add all necessary component registrations. If you are registering IDisposable components that need to be created and destroyed on a per client basis (i.e. an EntityFramework DataContext), please ensure that you use the HierarchicalLifetimeManager:

```
container
  .RegisterType<IService1, Service1>()
  .RegisterType<IRespository<Blah>, BlahRepository>()
  .RegisterType<IBlahContext, BlahContext>(new HierarchicalLifetimeManager());
```

Do you need professional PDFs? Try **PDFmyURL!**

WINDOWS SERVICE HOSTING

If you are hosting your WCF service within a Windows Service using a ServiceHost, replace the ServiceHost instance with the custom Unity.Wcf.UnityServiceHost. You will find that the UnityServiceHost takes in a Unity container as its first parameter but is otherwise identical to the default ServiceHost.

Delete the WcfServiceFactory class that has been added to the root of your application. It is not necessary for non-WAS hosting. Instead, you are free to configure Unity any way you like as long as the configured container is passed into the UnityServiceHost correctly. As with WAS hosting, if you want Unity.WCF to dispose of IDisposable components, you must register those components using the HierarchicalLifetimeManager lifestyle.

MY IDISPOSABLE INSTANCES ARE NOT BEING DISPOSED

If you add the Unity.WCF NuGet package to your application and get everything up and running and your disposable object are still not being disposed...then you are probably doing something wrong. The most likely causes are:

(1) You did not register your disposable type correctly with Unity. Remember than you must specify the HierarchicalLifetimeManager.

(2) Your client is not closing the connection to your service and you are using InstanceContextMode.PerSession (the default). Make sure that your client always closes the channel once it has finished calling it. Also consider changing the context mode to PerCall to remove this client responsibility and also make your service far more scalable.

CONCLUSION

Unity.WCF is a small library that allows simple integration of the Unity IoC container into WCF projects. Unlike many online examples, Unity.WCF will automatically dispose of configured IDisposable instances at the end of each request making it particularly useful when using ORM's such as Entity Framework. Unity.WCF is now available as a NuGet package and as a full source download on **CodePlex**.

Do you need professional PDFs? Try **PDFmyURL!**

USEFUL OR INTERESTING?

If you liked the article, we would really appreciate it if you would share it with your Twitter followers.

SHARE ON TWITTER »

COMMENTS

SHOW COMMENTS »

Comments are now closed for this article.

© 2014 DevTrends. All Rights Reserved.

Do you need professional PDFs? Try **PDFmyURL!**