# Caliburn Micro:

A Micro-Framework for WPF, Silverlight and WP7

### Collected Documentation

This document is a collection of the content located at <http://caliburnmicro.codeplex.com/documentation>. The content is current as of October 14, 2010.

Caliburn.Micro is licensed under the MIT License

Copyright (c) 2010 Blue Spire Consulting, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Documentation

## Introduction

When my [“Build Your Own MVVM Framework”](http://live.visitmix.com/MIX10/Sessions/EX15) talk was chosen for Mix10, I was excited to have the opportunity to show others what we had been doing in [Caliburn](http://caliburn.codeplex.com/) in a simplified, but powerful way. After giving the talk, I received a ton of positive feedback on the techniques/framework that I had demonstrated. I was approached by several companies and individuals who expressed an interest in a more “official” version of what I had shown. That, coupled with the coming of Windows Phone 7, impressed upon me a need to have a more “lean and mean” framework. My vision was to take 90% of Caliburn’s features and squash them into 10% of the code. I also hoped I could fit it in about 50k, give or take. I started with the Mix sample framework, then used an iterative process of integrating Caliburn v2 features, simplifying and refactoring. I continued along those lines until I had what I felt was a complete solution that mirrored the full version of Caliburn v2, but on a smaller scale.  
  
Caliburn.Micro consists of one ~50k assembly that builds for WPF4, SL4 and WP7. It has a single dependency, System.Windows.Interactivity, which you are probably already using regularly in development. Caliburn.Micro is about 2,000 LOC total, so you can easily go through the whole codebase in an afternoon and hold it in your head. That’s less than 10% of the size of Caliburn v2, which is running around 27,000 LOC and is a lot harder to grasp in a short time. So, it looks like I was able to meat my code quantity and size goals. But the best part is that I believe I was able to put something together that contains all of the features I consider most important in Caliburn. Here’s a brief list:

* **ActionMessages** – The Action mechanism allows you to “bind” UI triggers, such as a Button’s “Click” event, to methods on your View-Model or Presenter. The mechanism allows for passing parameters to the method as well. Parameters can be databound to other FrameworkElements or can pass special values, such as the DataContext or EventArgs. All parameters are automatically type converted to the method’s signature. This mechanism also allows the “Action.Target” to vary independently of the DataContext and enables it to be declared at different points in the UI from the trigger. When a trigger occurs, the “message” bubbles through the element tree looking for an Action.Target (handler) that is capable of invoking the specified method. This is why we call them messages. The “bubbling” nature of Acton Messages is extremely powerful and very helpful especially in master/detail scenarios. In addition to invocation, the mechanism supports a “CanExecute” guard. If the Action has a corresponding Property or Method with the same name, but precede by the word “Can,” the invocation of the Action will be blocked and the UI will be disabled. Actions also support Coroutines (see below). That’s all fairly standard for existing Caliburn users, but we do have a few improvements in Caliburn.Micro that will be making their way into the larger framework. The Caliburn.Micro implementation of ActionMessages is built on System.Windows.Interactivity. This allows actions to be triggered by any TriggerBase developed by the community. Furthermore, Caliburn.Micro’s Actions have full design-time support in Blend. Code-centric developers will be happy to know that Caliburn.Micro supports a very terse syntax for declaring these ActionMessages through a special attached property called Message.Attach.
* **Action Conventions** – Out of the box, we support a set of binding conventions around the ActionMessage feature. These conventions are based on x:Name. So, if you have a method called “Save” on your ViewModel and a Button named “Save” in your UI, we will automatically create an EventTrigger for the “Click” event and assign an ActionMessage for the “Save” method. Furthermore, we will inspect the method’s signature and properly construct the ActionMessage parameters. This mechanism can be turned off or customized. You can even change or add conventions for different controls. For example, you could make the convention event for Button “MouseMove” instead of “Click” if you really wanted.
* **Binding Conventions** – We also support convention-based databinding. This too works with x:Name. If you have a property on your ViewModel with the same name as an element, we will attempt to databind them. Whereas the framework understands convention events for Actions, it additionally understands convention binding properties (which you can customize or extend). When a binding name match occurs, we then proceed through several steps to build up the binding (all of which are customizable), configuring such details as BindingMode, StringFormat, ValueConverter, Validation and UpdateSourceTrigger (works for SL TextBox and PasswordBox too). Finally, we support the addition of custom behaviors for certain scenarios. This allows us to detect whether we need to auto-generate a DataTemplate or wire both the ItemsSource *and* the SelectedItem of a Selector based on naming patterns.
* **Screens and Conductors** – The Screen, ScreenConductor and ScreenCollection patterns enable model-based tracking of the active or current item, enforcing of screen lifecycles and elegant shutdown or shutdown cancellation in an application. Caliburn.Micro’s implementation of these patterns is an evolution of the one found in Caliburn and supports conducting any type of class, not just implementations of IScreen. These improvements are being introduced back into Caliburn. You’ll find that Caliburn.Micro’s screen implementation is quite thorough and even handles asynchronous shutdown scenarios with ease.
* **Event Aggregator** – Coming in at about 75LOC, Caliburn.Micro’s EventAggregator is simple yet powerful. The aggregator follows a bus-style pub/sub model. You register a message handler with the aggregator, and it sends you any messages you are interested in. You declare your interest in a particular message type by implementing IHandle<TMessage>. References to handlers are held weakly and publication occurs on the UI thread.
* **Coroutines** – Any action can optionally choose to return IResult or IEnumerable<IResult>, opening the door to a powerful approach to handling asynchronous programming. Furthermore, implementations of IResult have access to an execution context which tells them what ActionMessage they are executing for, what FrameworkElement triggered the messsage to be sent, what instance the ActionMessage was handled by (invoked on) and what the View is for that instance. Such contextual information enables a loosely-coupled, declarative mechanism by which a Presenter or View-Model can communicate with it’s View without needing to hold a reference to it at any time.
* **ViewLocator** – For every ViewModel in your application, Caliburn.Micro has a basic strategy for locating the View that should render it. We do this based on naming conventions. For example, if your VM is called MyApplication.ViewModels.ShellViewModel, we will look for MyApplication.Views.ShellView. Additionally, we support multiple views over the same View-Model be attaching a View.Context in Xaml. So, given the same model as above, but with a View.Context=”Master” we would search for MyApplication.Views.Shell.Master. Of coarse, all this is customizable.
* **WindowManager** – This service provides a View-Model-centric way of displaying Windows (ChildWindow in SL and Window in WPF). Simply pass it an instance of the VM and it will locate the view, wrap it in a Window if necessary, apply all conventions you have configured and show the window.
* **PropertyChangedBase and BindableCollection** – What self respecting WPF/SL framework could go without a base implementation of INotifyPropertyChanged? The Caliburn.Micro implementation enables string and lambda-based change notification. It also ensures that all events are raised on the UI thread. BindableCollection is a simple collection that inherits from ObservableCollection<T>, but that ensures that all its events are raised on the UI thread as well.
* **Bootstrapper** – What’s required to configure this framework and get it up and running? Not much. Simply inherit from Bootsrapper and add an instance of your custom bootstrapper to the Application’s ResourceDictionary. Done. If you want, you can override a few methods to plug in your own IoC container, declare what assemblies should be inspected for Views, etc. It’s pretty simple.
* **Logging** – Caliburn.Micro implements a basic logging abstraction. This is important in any serious framework that encourages Convention over Configuration. All the most important parts of the framework are covered with logging. Want to know what conventions are or are not being applied? Turn on logging. Want to know what actions are being executed? Turn on logging. Want to know what events are being published? Turn on logging. You get the picture.
* **MVVM and MVP** – In case it isn’t obvious, this framework enables MVVM. MVVM isn’t hard on it’s own, but Caliburn.Micro strives to go beyond simply getting it done. We want to write elegant, testable, maintainable and extensible presentation layer code…and we want it to be easy to do so. That’s what this is about. If you prefer using Supervising Controller and PassiveView to MVVM, go right ahead. You’ll find that Caliburn.Micro can help you a lot, particularly it’s Screen/ScreenConductor implementation. If you are not interested in any of the goals I just mentioned, you’d best move along. This framework isn’t for you.

Just to be clear, this isn’t a toy framework. As I said, I really focused on supporting the core and most commonly used features from Caliburn v2. In fact, Caliburn.Micro is going to be my default framework moving forward and I recommend that if you are starting a new project you begin with the Micro framework. I’ve been careful to keep the application developer API consistent with the full version of Caliburn. In fact, the improvements I made in Caliburn.Micro are being folded back into Caliburn v2. What’s the good part about that? You can start developing with Caliburn.Micro, then if you hit edge cases or have some other need to move to Caliburn, you will be able to do so with little or no changes in your application.  
  
Keeping with the spirit of “Build Your Own…” I want developers to understand how this little framework works, inside and out. I’ve intentionally chosen Mercurial for source control, because I want developers to take ownership. While I’ve done some work to make the most important parts of the framework extensible, I’m hoping to see many Forks, each with their own customizations *unique to their application’s needs.*

## Obtain the Code

There are two ways to get the code:

### Mercurial

Caliburn.Micro uses Mercurial for source control. There are many Mercurial clients available that can be used with CodePlex. TortoiseHG is a popular Mercurial client that works as an extension in Windows Explorer. For more information please see [Using TortoiseHG with CodePlex](http://codeplex.codeplex.com/wikipage?title=Using%20TortoiseHG%20with%20CodePlex&referringTitle=Source%20control%20clients&ProjectName=codeplex). If you are new to Mercurial, I highly recommend that you watch the free TekPub video on the subject [here](http://tekpub.com/codeplex).  
  
**Clone URL:** https://hg01.codeplex.com/caliburnmicro  
**Username:** <your codeplex user name>  
**Password:** <same as your codeplex password>

### Direct download

I only recommend this method if you are really uncomfortable with Mercurial. Simply navigate to the Source Code tab, click the link for the latest revision, then click the download link.

### Build the Code

1. Navigate to the folder where you downloaded the source code.
2. Navigate to the "src" directory.
3. Open Caliburn.Micro.sln.
4. Press Ctrl-Shift-B (or use the Build menu) to build the solution.
5. Locate the assemblies in the "bin" directory under the appropriate platform's project folder.

## Basic Configuration, Actions and Conventions

Open Visual Studio and create a new Silverlight 4 Application called “Caliburn.Micro.Hello”. You don’t need a web site or test project. Add a reference to System.Windows.Interactivity.dll and Caliburn.Micro.dll. You can find them both in the \src\Caliburn.Micro.Silverlight\Bin\Release (or Debug) folder. Delete “MainPage.xaml” and clean up your “App.xaml.cs” so that it looks like this:

namespace Caliburn.Micro.Hello

{

using System.Windows;

public partial class App : Application

{

public App()

{

InitializeComponent();

}

}

}

Since Caliburn.Micro prefers a View-Model-First approach, let’s start there. Create your first VM and call it ShellViewModel. Use the following code for the implementation:

namespace Caliburn.Micro.Hello

{

using System.Windows;

public class ShellViewModel : PropertyChangedBase

{

string name;

public string Name

{

get { return name; }

set

{

name = value;

NotifyOfPropertyChange(() => Name);

NotifyOfPropertyChange(() => CanSayHello);

}

}

public bool CanSayHello

{

get { return !string.IsNullOrWhiteSpace(Name); }

}

public void SayHello()

{

MessageBox.Show(string.Format("Hello {0}!", Name)); //Don't do this in real life :)

}

}

}

Notice that the ShellViewModel inherits from PropertyChangedBase. This is a base class that implements the infrastructure for property change notification and automatically performs UI thread marshalling. It will come in handy :)  
  
Now that we have our VM, let’s create the bootstrapper that will configure the framework and tell it what to do. Create a new class named HelloBootstrapper. You can use this tiny bit of code:

namespace Caliburn.Micro.Hello

{

public class HelloBootstrapper : Bootstrapper<ShellViewModel> {}

}

There are two Bootsrappers available in Caliburn.Micro. This version allows you to specify the type of “root view model” via the generic type. The “root view mdoel” is a VM that Caliburn.Micro will instantiate and use to show your application. Next, we need to place the HelloBootstrapper somewhere where it will be run at startup. To do that, change your App.xaml to match this:  
  
*Silverlight*

<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Caliburn.Micro.Hello"

x:Class="Caliburn.Micro.Hello.App">

<Application.Resources>

<local:HelloBootstrapper x:Key="bootstrapper" />

</Application.Resources>

</Application>

*WPF*

<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Caliburn.Micro.Hello"

x:Class="Caliburn.Micro.Hello.App">

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary>

<local:HelloBootstrapper x:Key="bootstrapper" />

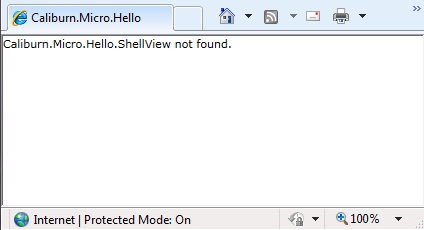
</ResourceDictionary>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

</Application>

All we have to do is place a Caliburn.Micro bootstrapper in the Application.Resources and it will do the rest of the work.\* Now, run the application. You should see something like this:  
  
  
  
Caliburn.Micro creates the ShellViewModel, but doesn’t know how to render it. So, let’s create a view. Create a new Silverlight User Control named ShellView. Use the following xaml:

<UserControl x:Class="Caliburn.Micro.Hello.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<StackPanel>

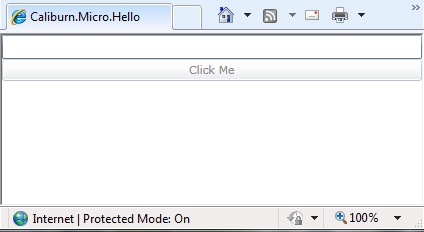
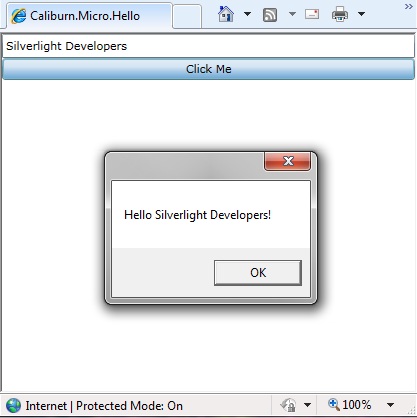
<TextBox x:Name="Name" />

<Button x:Name="SayHello"

Content="Click Me" />

</StackPanel>

</UserControl>

Run the application again. You should now see the UI:  
  
  
  
Typing something in the TextBox will enable the Button and clicking it will show a message:  
  
  
  
Caliburn.Micro uses a simple naming convention to locate Views for ViewModels. Essentially, it takes the FullName and removes “Model” from it. So, given MyApp.ViewModels.MyViewModel, it would look for MyApp.Views.MyView. Looking at the View and ViewModel side-by-side, you can see that the TextBox with x:Name=”Name” is bound to the “Name” property on the VM. You can also see that the Button with x:Name=”SayHello” is bound to the method with the same name on the VM. The “CanSayHello” property is guarding access to the “SayHello” action by disabling the Button. These are the basics of Caliburn.Micro’s ActionMessage and Conventions functionality. There’s much more to show. But, next time I want to show how we can integrate an IoC container such as MEF.

**Samples**

[Caliburn Micro Hello](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=130835)  
  
**Footnotes**

1. All the functionality described in this article works identically for both Caliburn.Micro and Caliburn…with one exception. Caliburn does not have a Bootstrapper currently. At this time, you would inherit from CaliburnApplication and override CreateRootModel. In the coming months, Caliburn’s custom application class will be replaced by a bootstrapper mechanism similar to Caliburn.Micro.

## Customizing The Bootstrapper

[In the last part](http://caliburnmicro.codeplex.com/wikipage?title=Basic%20Configuration%2c%20Actions%20and%20Conventions&referringTitle=Customizing%20The%20Bootstrapper) we discussed the most basic configuration for Caliburn.Micro and demonstrated a couple of simple features related to Actions and Conventions. In this part, I would like to explore the Bootstrapper class a little more. Let’s begin by configuring our application to use an IoC container. We’ll use MEF for this example, but Caliburn.Micro will work well with any container. First, go ahead and grab the code from Part 1 [or download the code for this article](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=131426). We are going to use that as our starting point. Add two additional references: System.ComponentModel.Composition and System.ComponentModel.Composition.Initialization. Those are the assemblies that contain MEF’s functionality.\* Now, let’s create a new Bootstrapper called MefBootstrapper. Use the following code:

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

using System.ComponentModel.Composition.Primitives;

using System.Linq;

public class MefBootstrapper : Bootstrapper<IShell>

{

private CompositionContainer container;

protected override void Configure()

{

container = CompositionHost.Initialize(

new AggregateCatalog(

AssemblySource.Instance.Select(x => new AssemblyCatalog(x)).OfType<ComposablePartCatalog>()

)

);

var batch = new CompositionBatch();

batch.AddExportedValue<IWindowManager>(new WindowManager());

batch.AddExportedValue<IEventAggregator>(new EventAggregator());

batch.AddExportedValue(container);

container.Compose(batch);

}

protected override object GetInstance(Type serviceType, string key)

{

string contract = string.IsNullOrEmpty(key) ? AttributedModelServices.GetContractName(serviceType) : key;

var exports = container.GetExportedValues<object>(contract);

if (exports.Count() > 0)

return exports.First();

throw new Exception(string.Format("Could not locate any instances of contract {0}.", contract));

}

protected override IEnumerable<object> GetAllInstances(Type serviceType)

{

return container.GetExportedValues<object>(AttributedModelServices.GetContractName(serviceType));

}

protected override void BuildUp(object instance)

{

container.SatisfyImportsOnce(instance);

}

}

That’s all the code to integrate MEF. First, we override the Configure method of the Bootstrapper class. This gives us an opportunity to set up our IoC container as well as perform any other framework configuration we may want to do, such as customizing conventions. In this case, I’m taking advantage of Silverlight’s CompositionHost to setup the CompositionContainer. You can just instantiate the container directly if you are working with .NET. Then, I’m creating an AggregateCatalog and populating it with AssemblyCatalogs; one for each Assembly in AssemblySource.Instance. So, what is AssemblySoure.Instance? This is the place that Caliburn.Micro looks for Views. You can add assemblies to this at any time during your application to make them available to the framework, but there is also a special place to do it in the Bootstrapper. Simply override SelectAssemblies like this:

protected override IEnumerable<Assembly> SelectAssemblies()

{

return new[] {

Assembly.GetExecutingAssembly()

};

}

All you have to do is return a list of searchable assemblies. By default, the base class returns the assembly that your Application exists in. So, if all your views are in the same assembly as your application, you don’t even need to worry about this. If you have multiple referenced assemblies that contain views, this is an extension point you need to remember. Also, if you are dynamically loading modules, you’ll need to make sure they get registered with your IoC container and the AssemblySoure.Instance when they are loaded.   
  
After creating the container and providing it with the catalogs, I make sure to add a few Caliburn.Micro-specific services. The framework provides default implementations of both IWindowManager and IEventAggregator. Those are pieces that I’m likely to take dependencies on elsewhere, so I want them to be available for injection. I also register the container with itself (just a personal preference).  
  
After we configure the container, we need to tell Caliburn.Micro how to use it. That is the purpose of the three overrides that follow. “GetInstance” and “GetAllInstances” are required by the framework. “BuildUp” is optionally used to supply property dependencies to instances of IResult that are executed by the framework.   
  
  
**Word to the Wise**  
  
*While Caliburn.Micro does provide ServiceLocator functionality through the Bootstrapper’s overrides and the IoC class, you should avoid using this directly in your application code. ServiceLocator is considered by many to be an anti-pattern. Pulling from a container tends to obscure the intent of the dependent code and can make testing more complicated. In future articles I will demonstrate at least one scenario where you may be tempted to access the ServiceLocator from a ViewModel. I’ll also demonstrate some solutions.*  
  
Besides what is shown above, there are some other notable methods on the Bootstrapper. You can override OnStartup and OnExit to execute code when the application starts or shuts down respectively and OnUnhandledException to cleanup after any exception that wasn’t specifically handled by your application code. The last override, DisplayRootView, is unique. Let’s look at how it is implemented in Bootstrapper<TRootModel>

protected override void DisplayRootView()

{

var viewModel = IoC.Get<TRootModel>();

#if SILVERLIGHT

var view = ViewLocator.LocateForModel(viewModel, null, null);

ViewModelBinder.Bind(viewModel, view, null);

var activator = viewModel as IActivate;

if (activator != null)

activator.Activate();

Application.RootVisual = view;

#else

IWindowManager windowManager;

try

{

windowManager = IoC.Get<IWindowManager>();

}

catch

{

windowManager = new WindowManager();

}

windowManager.Show(viewModel);

#endif

}

The Silverlight version of this method resolves your root VM from the container, locates the view for it and binds the two together. It then makes sure to “activate” the VM if it implements the appropriate interface. The WPF version does the same thing by using the WindowManager class, more or less. DisplayRootView is basically a convenience implementation for model-first development. If you don’t like it, perhaps because you prefer view-first MVVM, then this is the method you want to override to change that behavior.  
  
Now that you understand all about the Bootstrapper, let’s get our sample working. We need to add the IShell interface. In our case, it’s just a marker interface. But, in a real application, you would have some significant shell-related functionality baked into this contract. Here’s the code:

public interface IShell

{

}

Now, we need to implement the interface and decorate our ShellViewModel with the appropriate MEF attributes:

[Export(typeof(IShell))]

public class ShellViewModel : PropertyChangedBase, IShell

{

...implementation is same as before...

}

That’s it! Your up and running with MEF and you have a handle on some of the other key extension point of the Bootstrapper as well.

**Samples**

[Hello Mef](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=131426)  
  
**Footnotes**

1. If you are using .NET, you will only need to reference System.ComponentModel.Composition.
2. I’m quite guilty of this myself, but I’m trying to be more conscious of it. I’m also excited to see that modern IoC containers as well as Caliburn.Micro provide some very nice ways to avoid this situation.

## All About Actions

We briefly introduced actions in [Pt. 1](http://caliburnmicro.codeplex.com/wikipage?title=Basic%20Configuration%2c%20Actions%20and%20Conventions&referringTitle=All%20About%20Actions), but there is so much more to know. To begin our investigation, we’ll take our simple “Hello” example and see what it looks like when we explicitly create the actions rather than use conventions. Here’s the Xaml:

<UserControl x:Class="Caliburn.Micro.Hello.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"

xmlns:cal="http://www.caliburnproject.org">

<StackPanel>

<TextBox x:Name="Name" />

<Button Content="Click Me">

<i:Interaction.Triggers>

<i:EventTrigger EventName="Click">

<cal:ActionMessage MethodName="SayHello" />

</i:EventTrigger>

</i:Interaction.Triggers>

</Button>

</StackPanel>

</UserControl>

As you can see, the Actions feature leverages System.Windows.Interactivity for it’s trigger mechanism. This means that you can use anything that inherits from System.Windows.Interactivity.TriggerBase to trigger the sending of an ActionMessage.\* Perhaps the most common trigger is an EventTrigger, but you can create almost any kind of trigger imaginable or leverage some common triggers already created by the community. ActionMessage is, of course, the Caliburn.Micro-specific part of this markup. It indicates that when the trigger occurs, we should send a message of “SayHello.” So, why do I use the language “send a message” instead of “execute a method” when describing this functionality? That’s the interesting and powerful part. ActionMessage bubbles through the Visual Tree searching for a target instance that can handle it. If a target is found, but does not have a “SayHello” method, the framework will continue to bubble until it finds one, throwing an exception if no “handler” is found.\*\* This bubbling nature of ActionMessage comes in handy in a number of interesting scenarios, Master/Details being a key use case. Another important feature to note is Action guards. When a handler is found for the “SayHello” message, it will check to see if that class also has either a property or a method named “CanSayHello.” If you have a guard property and your class implements INotifyPropertyChanged, then the framework will observe changes in that property and re-evaluate the guard accordingly. We’ll discuss method guards in further detail below.

### Action Targets

Now you’re probably wondering how to specify the target of an ActionMessage. Looking at the markup above, there’s no visible indication of what that target will be. So, where does that come from? Since we used a Model-First approach, when Caliburn.Micro (hereafter CM) created the view and bound it to the ViewModel using the ViewModelBinder, it set this up for us. Anything that goes through the ViewModelBinder will have its action target set automatically. But, you can set it yourself as well, using the attached property Action.Target. Setting this property positions an ActionMessage “handler” in the Visual Tree attached to the node on with you declare the property. It also sets the DataContext to the same value, since you often want these two things to be the same. However, you can vary the Action.Target from the DataContext if you like. Simply use the Action.TargetWithoutContext attached property instead. One nice thing about Action.Target is that you can set it to a System.String and CM will use that string to resolve an instance from the IoC container using the provided value as its key. This gives you a nice way of doing View-First MVVM if you so desire. If you want Action.Target set and you want Action/Binding Conventions applied as well, you can use the Bind.Model attached property in the same way.

### View First

Let’s see how we would apply this to achieve MVVM using a View-First technique (gasp!) Here’s how we would change our bootstrapper:

public class MefBootstrapper : Bootstrapper

{

//same as before

protected override void DisplayRootView()

{

Application.Current.RootVisual = new ShellView();

}

//same as before

}

Because we are using View-First, we’ve inherited from the non-generic Bootstrapper. The MEF configuration is the same as seen previously, so I have left that out for brevity’s sake. The only other thing that is changed is how the view gets created. In this scenario, we simply override DisplayRootView, instantiate the view ourselves and set it as the RootVisual (or call Show in the case of WPF). Next, we’ll slightly alter how we are exporting our ShellViewModel, by adding an explicitly named contract:

[Export("Shell", typeof(IShell))]

public class ShellViewModel : PropertyChangedBase, IShell

{

//same as before

}

Finally, we will alter our view to pull in the VM and perform all bindings:

<UserControl x:Class="Caliburn.Micro.ViewFirst.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:cal="http://www.caliburnproject.org"

cal:Bind.Model="Shell">

<StackPanel>

<TextBox x:Name="Name" />

<Button x:Name="SayHello"

Content="Click Me" />

</StackPanel>

</UserControl>

Notice the use of the Bind.Model attached property. This resolves our VM by key from the IoC container, sets the Action.Target and DataContext and applies all conventions. I thought it would be nice to show how View-First development is fully supported with CM, but mainly I want to make clear the various ways that you can set targets for actions and the implications of using each technique. Here’s a summary of the available attached properties:

* **Action.Target** – Sets both the Action.Target property and the DataContext property to the specified instance. String values are used to resolve an instance from the IoC container.
* **Action.TargetWithoutContext** – Sets only the Action.Target property to the specified instance. String values are used to resolve an instance from the IoC container.
* **Bind.Model** – View-First - Set’s the Action.Target and DataContext properties to the specified instance. Applies conventions to the view. String values are used to resolve an instance from the IoC container.
* **View.Model** – ViewModel-First – Locates the view for the specified VM instance and injects it at the content site. Sets the VM to the Action.Target and the DataContext. Applies conventions to the view.

### Action Parameters

Now, let’s take a look at another interesting aspect of ActionMessage: Parameters. To see this in action, let’s switch back to our original ViewModel-First bootstrapper, etc. and begin by changing our ShellViewModel to look like this:

using System.ComponentModel.Composition;

using System.Windows;

[Export(typeof(IShell))]

public class ShellViewModel : IShell

{

public bool CanSayHello(string name)

{

return !string.IsNullOrWhiteSpace(name);

}

public void SayHello(string name)

{

MessageBox.Show(string.Format("Hello {0}!", name));

}

}

There are a few things to note here. First, we are now working with a completely POCO class; no INPC goop here. Second, we have added an input parameter to our SayHello method. Finally, we changed our CanSayHello property into a method with the same inputs as the action, but with a bool return type. Now, let’s have a look at the Xaml:

<UserControl x:Class="Caliburn.Micro.HelloParameters.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"

xmlns:cal="http://www.caliburnproject.org">

<StackPanel>

<TextBox x:Name="Name" />

<Button Content="Click Me">

<i:Interaction.Triggers>

<i:EventTrigger EventName="Click">

<cal:ActionMessage MethodName="SayHello">

<cal:Parameter Value="{Binding ElementName=Name, Path=Text}" />

</cal:ActionMessage>

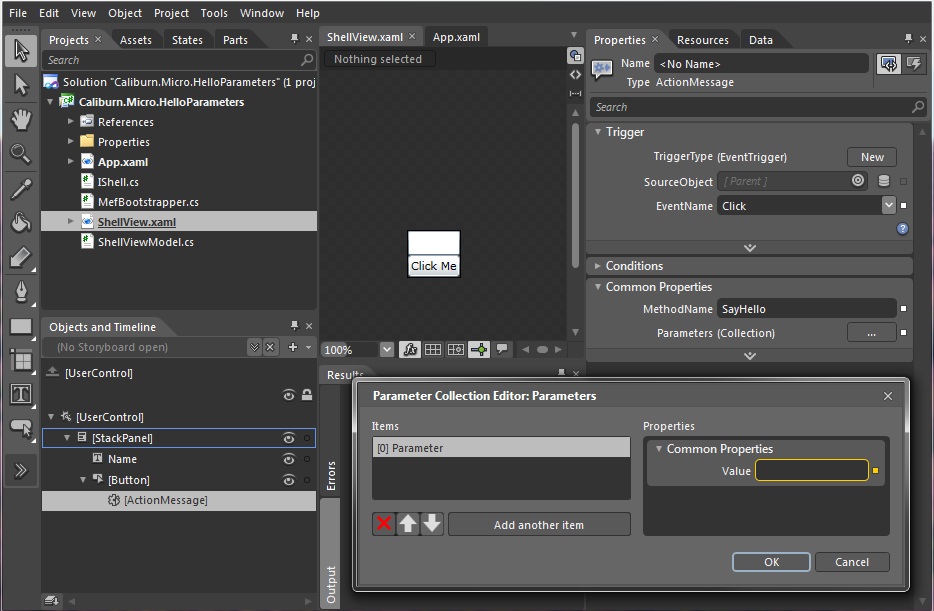
</i:EventTrigger>

</i:Interaction.Triggers>

</Button>

</StackPanel>

</UserControl>

Our markup now has one modification: We declared the parameter as part of the ActionMessage using an ElementName Binding. You can have any number of parameters you desire. Value is a DependencyProperty, so all the standard binding capabilities apply to parameters. Did I mention you can do all this in Blend?  
  
  
  
  
  
One thing that is nice about this is that every time the value of a parameter changes, we’ll call the guard method associated with the action(CanSayHello in this case) and use its result to update the UI that the ActionMessage is attached to. Go ahead and run the application. You’ll see that it behaves the same as in previous examples.  
  
In addition to literal values and Binding Expressions, there are a number of helpful “special” values that you can use with parameters. These allow you a convenient way to access common contextual information:

* **$eventArgs** – Passes the Trigger’s EventArgs or input parameter to your Action. Note: This will be null for guard methods since the trigger hasn’t actually occurred.
* **$dataContext** – Passes the DataContext of the element that the ActionMessage is attached to. This is very useful in Master/Detail scenarios where the ActionMessage may bubble to a parent VM but needs to carry with it the child instance to be acted upon.
* **$source** – The actual FrameworkElement that triggered the ActionMessage to be sent.

You must start the variable with a “$” but the name is treated in a case-insensitive way by CM.  
  
  
  
**Word to the Wise**  
  
*Parameters are a convenience feature. They are very powerful and can help you out of some tricky spots, but they can be easily abused. Personally, I only use parameters in the simplest scenarios. One place where they have worked nicely for me is in login forms. Another scenario, as mentioned previously is Master/Detail operations.*  
  
  
  
Now, do you want to see something truly wicked? Change your Xaml back to this:

<UserControl x:Class="Caliburn.Micro.HelloParameters.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<StackPanel>

<TextBox x:Name="Name" />

<Button x:Name="SayHello"

Content="Click Me" />

</StackPanel>

</UserControl>

Running the application will confirm for you that CM’s conventions even understand ActionMessage parameters. We’ll discuss conventions a lot more in the future, but you should be happy to know that these conventions are case-insensitive and can even detect the before-mentioned “special” values.

### Action Bubbling

Now, lets look at a simple Master/Detail scenario that demonstrates ActionMessage bubbling, but let’s do it with a shorthand syntax that is designed to be more developer friendly. We’ll start by adding a simple new class named Model:

using System;

public class Model

{

public Guid Id { get; set; }

}

And then we’ll change our ShellViewModel to this:

using System;

using System.ComponentModel.Composition;

[Export(typeof(IShell))]

public class ShellViewModel : IShell

{

public BindableCollection<Model> Items { get; private set; }

public ShellViewModel()

{

Items = new BindableCollection<Model>{

new Model { Id = Guid.NewGuid() },

new Model { Id = Guid.NewGuid() },

new Model { Id = Guid.NewGuid() },

new Model { Id = Guid.NewGuid() }

};

}

public void Add()

{

Items.Add(new Model { Id = Guid.NewGuid() });

}

public void Remove(Model child)

{

Items.Remove(child);

}

}

Now our shell has a collection of Model instances along with the ability to add or remove from the collection. Notice that the Remove method takes a single parameter of type Model. Now, let’s update the ShellView:

<UserControl x:Class="Caliburn.Micro.BubblingAction.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:cal="http://www.caliburnproject.org">

<StackPanel>

<ItemsControl x:Name="Items">

<ItemsControl.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<Button Content="Remove"

cal:Message.Attach="Remove($dataContext)" />

<TextBlock Text="{Binding Id}" />

</StackPanel>

</DataTemplate>

</ItemsControl.ItemTemplate>

</ItemsControl>

<Button Content="Add"

cal:Message.Attach="Add" />

</StackPanel>

</UserControl>

### Message.Attach

The first thing to notice is that we are using a more Xaml-developer-friendly mechanism for declaring our ActionMessages. The Message.Attach property is backed by a simple parser which takes its textual input and transforms it into the full Interaction.Trigger/ActionMessage that you’ve seen previously. If you work primarily in the Xaml editor and not in the designer, you’re going to like Message.Attach. Notice that neither Message.Attach declarations specify which event should send the message. If you leave off the event, the parser will use the ConventionManager to determine the default event to use for the trigger. In the case of Button, it’s Click. You can always be explicit of coarse. Here’s what the full syntax for our Remove message would look like if we were declaring everything:

<Button Content="Remove"

cal:Message.Attach="[Event Click] = [Action Remove($dataContext)]" />

Suppose we were to re-write our parameterized SayHello action with the Message.Attach syntax. It would look like this:

<Button Content="Click Me"

cal:Message.Attach="[Event Click] = [Action SayHello(Name.Text)]" />

But we could also leverage some smart defaults of the parser and do it like this:

<Button Content="Click Me"

cal:Message.Attach="SayHello(Name)" />

You can specify literals as parameters as well and even declare multiple actions by separating them with a semicolon:

<Button Content="Let's Talk"

cal:Message.Attach="[Event MouseEnter] = [Action Talk('Hello', Name.Text)]; [Event MouseLeave] = [Action Talk('Goodbye', Name.Text)]" />

**WARNING** Those developers who ask me to expand this functionality into a full-blown expression parser will be taken out back and…dealt with. Message.Attach is not about cramming code into Xaml. It’s purpose is to provide a streamlined syntax for declaring when/what messages to send to the ViewModel. Please don’t abuse this.  
  
  
If you haven’t already, run the application. Any doubts you had will hopefully be put to rest when you see that the message bubbling works as advertised :) Something else I would like to point out is that CM automatically performs type-conversion on parameters. So, for example, you can pump TextBox.Text into a System.Double parameter without any fear of a casting issue.  
  
So, we’ve discussed using Interaction.Triggers with ActionMessage, including the use of Parameters with literals, element bindings\*\*\* and special values. We’ve discussed the various ways to set the action target depending on your needs/architectural style: Action.Target, Action.TargetWithoutContext, Bind.Model or View.Model. We also saw an example of the bubbling nature of ActionMessage and demoed it using the streamlined Message.Attach syntax. All along the way we’ve looked at various examples of conventions in action too. Now, there’s one final killer feature of ActionMessage we haven’t discussed yet…Coroutines. But, that will have to wait until next time.

### Samples

[Explicit Actions](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=133364)  
[View First](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=133365)  
[Parameters](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=133366)  
[Bubbling Actions](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=133367)  
  
  
**Footnotes**

1. Currently, the full version of Caliburn is not based on System.Windows.Interactivity. Caliburn’s trigger mechanism was around long before Blend’s. You may notice a shocking similarity in the markup. That said, Caliburn v2.0 will be migrated to use the Blend model in the near future.
2. Actually, if no handler is found, before an exception is thrown, the framework will check the current DataContext to see if it has the requested method. This seamed like a reasonable fallback behavior.
3. One important detail about ElementName Bindings that I didn’t mention…It doesn’t work with WP7 currently. Due to the fact that WP7 is based on a version of Silverlight 3 which had an incomplete implementation of DependencyObject/DependencyProperty, the infrastructure is not present to make this work in any sort of sane way. However, parameter literals and special values still work as described along with all the rest of the ActionMessage features.

## Working with Windows Phone 7

Hopefully, previous articles have you up to speed on what Caliburn.Micro is, its basic configuration, and how to take advantage of a few of its features. In this part, I want to talk about some WP7 specifics issues. It’s unfortunate that I have to call out WP7, but your going to find that while you may be an experienced WPF or Silverlight developer, that doesn’t make WP7 development a snap. Microsoft still has a long ways to go in making “three screens and the cloud” a reality. The new features in Caliburn.Micro are specifically designed to address some of the shortcomings in WP7, particularly around Navigation (with ViewModels and Screen Activation), Tombstoning and Launchers/Choosers.  
  
**Bootstrapper**  
  
Let’s start by getting our application configured correctly. In previous parts we began by creating a bootstrapper and adding it to our Application.Resources. We do the same thing on WP7, but we inherit our bootstrapper from the PhoneBootstrapper class. Here’s how the bootstrapper for our WP7 sample application looks:

using System;

using System.Collections.Generic;

using Microsoft.Phone.Tasks;

public class HelloWP7Bootstrapper : PhoneBootstrapper

{

PhoneContainer container;

protected override void Configure()

{

container = new PhoneContainer();

container.RegisterSingleton(typeof(MainPageViewModel), "MainPageViewModel", typeof(MainPageViewModel));

container.RegisterSingleton(typeof(PageTwoViewModel), "PageTwoViewModel", typeof(PageTwoViewModel));

container.RegisterPerRequest(typeof(TabViewModel), null, typeof(TabViewModel));

container.RegisterInstance(typeof(INavigationService), null, new FrameAdapter(RootFrame));

container.RegisterInstance(typeof(IPhoneService), null, new PhoneApplicationServiceAdapter(PhoneService));

container.Activator.InstallChooser<PhoneNumberChooserTask, PhoneNumberResult>();

container.Activator.InstallLauncher<EmailComposeTask>();

}

protected override object GetInstance(Type service, string key)

{

return container.GetInstance(service, key);

}

protected override IEnumerable<object> GetAllInstances(Type service)

{

return container.GetAllInstances(service);

}

protected override void BuildUp(object instance)

{

container.BuildUp(instance);

}

}

For WP7, there are less IoC container options available. In this case, I’ve written a simple container myself. You can get the full code for SimpleContainer in the Recipes section of the CM Documentation. Additionally, I inherited from SimpleContainer to create PhoneContainer, which just adds some custom activation logic related to Launchers/Choosers. We’ll dig into that a bit later. The most important thing to note in this code is the use of PhoneBootstrapper and the registrations of two services: INavigationService, which wraps the RootFrame and IPhoneService, which wraps the native PhoneService. The RootFrame and PhoneService are created by the PhoneBootstrapper and added to the appropriate places, so you can keep your App.xaml and App.xaml.cs files clean. Here’s what they should look like:

<Application x:Class="Caliburn.Micro.HelloWP7.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Caliburn.Micro.HelloWP7">

<Application.Resources>

<local:HelloWP7Bootstrapper x:Key="bootstrapper" />

</Application.Resources>

</Application>

using System.Windows;

public partial class App : Application

{

public App()

{

InitializeComponent();

}

}

**INavigationService**  
  
Let’s dig into what the FrameAdapter implementation of INavigationService does for you. First, you should know that WP7 enforces a View-First approach to UI at the platform level. Like it or not (I don’t) the platform is going to create pages at will and the Frame control is going to conduct your application thusly. You don’t get to control that and there are no extensibility points, unlike the Silverlight version of the navigation framework. Rather than fight this, I’m going to recommend embracing the View-First approach for Pages in WP7, but maintaining a Model-First composition strategy for the sub-components of those pages. In order to support the View-First approach, I’ve enabled the FrameAdapter to hook into the native navigation frame’s functionality and augment it with the following behaviors:  
  
*When Navigating To a Page*

* Use the new ViewModelLocator to conventionally determine the name of the VM that should be attached to the page being navigated to. Pull that VM by key out of the container.
* If a VM is found, use the ViewModelBinder to connect the Page to the located ViewModel.
* Examine the Page’s QueryString. Look for properties on the VM that match the QueryString parameters and inject them, performing the necessary type coercion.
* If the ViewModel implements the IActivate interface, call its Activate method.

*When Navigating Away From a Page*

* Detect whether the associated ViewModel implements the IGuardClose interface.
* If IGuardClose is implemented and the app is not being tombstoned or closed, invoke the CanClose method and use its result to optionally cancel page navigation.\*
* If the ViewModel can close and implements the IDeactivate interface, call it’s Deactivate method.
* If the application is being tombstoned or closed, pass “true” to the Deactivate method indicating that the ViewModel should permanently close.

The behavior of the navigation service allows the correct VM to be hooked up to the page, allows that VM to be notified that it is being navigated to (IActivate), allows it to prevent navigation away from the current page (IGuardClose) and allows it to clean up after itself on navigation away, tombstoning or normal “closing” of the application (IDeactivate). All these interfaces (and a couple more) are implemented by the Screen class. I haven’t discussed Screens and Conductors yet, but you can get started taking advantage of them with just the simple information I have provided. If you prefer not to inherit from Screen, you can implement any of the interfaces individually of coarse. They provide a nice View-Model-Centric, testable and predictable way of responding to navigation without needing to wire up a ton of event handlers or write important application flow logic in the page’s code-behind. Simply follow the same naming conventions for View/ViewModels as normal, plug the FrameAdapter into your IoC container and implement whatever interfaces you care about on your VMs.  
  
**IPhoneService**  
  
There’s really not much to say about this. I abstracted an interface for the built-in PhoneApplicationService and created a decorator. This makes it easy for VMs to take a dependency on the functionality without being coupled to the actual phone service. I didn’t add any additional behavior.  
  
  
Now that we have the basic service explanations over with, let’s create some Views and ViewModels to get our navigation working and to demonstrate a few other features. The default WP7 template creates a MainPage.xaml for you and configures the application to use that as its default starting point. We’ll use that, but change the Xaml to this:

<phone:PhoneApplicationPage x:Class="Caliburn.Micro.HelloWP7.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"

xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"

SupportedOrientations="Portrait"

Orientation="Portrait"

shell:SystemTray.IsVisible="True">

<Grid Background="Transparent">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<StackPanel Grid.Row="0"

Margin="24,24,0,12">

<TextBlock Text="WP7 Caliburn.Micro"

Style="{StaticResource PhoneTextNormalStyle}" />

<TextBlock Text="Main Page"

Margin="-3,-8,0,0"

Style="{StaticResource PhoneTextTitle1Style}" />

</StackPanel>

<Grid Grid.Row="1">

<Button x:Name="GotoPageTwo"

Content="Goto Page Two" />

</Grid>

</Grid>

</phone:PhoneApplicationPage>

The only noteworthy thing I have done here is to add a single Button to the page in order to demonstrate navigation. Here’s what the MainPageViewModel looks like:

using System;

public class MainPageViewModel

{

readonly INavigationService navigationService;

public MainPageViewModel(INavigationService navigationService)

{

this.navigationService = navigationService;

}

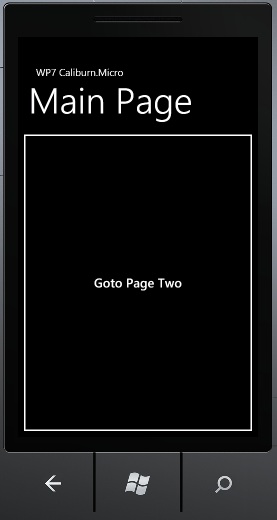
public void GotoPageTwo()

{

navigationService.Navigate(new Uri("/PageTwo.xaml?NumberOfTabs=5", UriKind.RelativeOrAbsolute));

}

}

Here’s what it would look like if we could run it at this point:  
  
  
  
It’s pretty simple, but there are a few things worth mentioning. The first is that our INavigationService is being injected through the constructor. This will happen when we navigate to the page and the VM is conventionally resolved and wired up. In this case we haven’t implemented any interfaces or inherited from any base classes. We don’t really care about lifecycle. Take a look at the GotoPageTwo method (conventionally wired to the big button). We are using the INavigationService to tell the phone to go to PageTwo.xaml. Remember the query string and have a look at the ViewModel for PageTwo:

using System;

using System.Collections.Generic;

using System.Linq;

[SurviveTombstone]

public class PageTwoViewModel : Conductor<IScreen>.Collection.OneActive

{

readonly Func<TabViewModel> createTab;

public PageTwoViewModel(Func<TabViewModel> createTab)

{

this.createTab = createTab;

}

public int NumberOfTabs { get; set; }

protected override void OnInitialize()

{

Enumerable.Range(1, NumberOfTabs).Apply(x =>{

var tab = createTab();

tab.DisplayName = "Item " + x;

Items.Add(tab);

});

ActivateItem(Items[0]);

}

}

The value of the the QueryString parameter “NumberOfTabs” is going to be injected into the property of the same name on the ViewModel. The OnInitialize method is part of the Activation lifecycle I mentioned above. If you notice in the bootstrapper’s configuration, PageTwoViewModel is a Singleton. OnInitialize will be called the first time this page is navigated to, not on any subsequent times, while OnActivate will be called every time this page is navigated to. These are properties of the Screen class from which this ViewModel ultimately inherits. Basically, all this method does is create a parameterized number of child ViewModels (TabViewModel) and then “Activates” the first one. The Items collection and Activation capabilities come from the Conductor base class. Just to remind you, I’m going to cover Screens and Conductors much more thoroughly in a future article. Also, take note of the SurviveTombstone attribute. We’ll get to that soon. First, here’s the view that this is bound to:

<phone:PhoneApplicationPage x:Class="Caliburn.Micro.HelloWP7.PageTwo"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"

xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"

SupportedOrientations="Portrait"

Orientation="Portrait"

shell:SystemTray.IsVisible="True">

<Grid Background="Transparent">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<StackPanel Grid.Row="0"

Margin="24,24,0,12">

<TextBlock Text="WP7 Caliburn.Micro"

Style="{StaticResource PhoneTextNormalStyle}" />

<TextBlock Text="Page Two"

Margin="-3,-8,0,0"

Style="{StaticResource PhoneTextTitle1Style}" />

</StackPanel>

<Grid Grid.Row="1">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<ListBox x:Name="Items">

<ListBox.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding DisplayName}"

Margin="2 0 12 0"

FontSize="30" />

</DataTemplate>

</ListBox.ItemTemplate>

<ListBox.ItemsPanel>

<ItemsPanelTemplate>

<StackPanel Orientation="Horizontal" />

</ItemsPanelTemplate>

</ListBox.ItemsPanel>

</ListBox>

<ContentControl x:Name="ActiveItem"

HorizontalContentAlignment="Stretch"

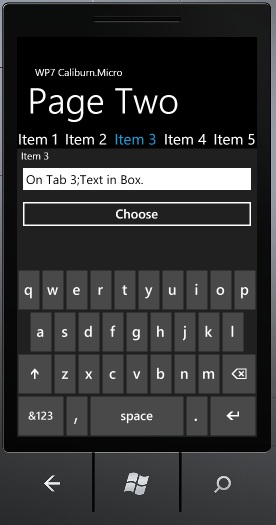
VerticalContentAlignment="Stretch"

Grid.Row="1" />

</Grid>

</Grid>

</phone:PhoneApplicationPage>

It looks like this when running:  
  
  
  
This view is a bit more complex, but you only need concern yourself with the ListBox and the ContentControl. Here, we are using these two controls to create a Tabbed UI. The ListBox plays the part of the TabHeaders. It’s name causes it to automatically have its ItemsSource bound to the Items collection on the VM and its SelectedItem bound to the ActiveItem on the VM (these properties come from Conductor base class). The ContentControl has its Content property automatically bound to the ActiveItem property on the View and it is done using the View.Model attached property I discussed in the previous article. This allows the ActiveItem to be conventionally wired up to it’s view, enabling the display of the “Tab Content”. And yes, I am going to cover conventions in more depth in a future article as well. But, I want you to get a taste of how conventions, composition and conductors all play together. What we’ve just created is an MDI interface inside this particular page completely driven by the ViewModel. Let’s take a look at that TabViewModel and it’s corresponding View. First the VM:

using System;

using System.Windows;

using Microsoft.Phone.Tasks;

[SurviveTombstone]

public class TabViewModel : Screen, ILaunchChooser<PhoneNumberResult>

{

string text;

[SurviveTombstone]

public string Text

{

get { return text; }

set

{

text = value;

NotifyOfPropertyChange(() => Text);

}

}

public void Choose()

{

TaskLaunchRequested(this, TaskLaunchEventArgs.For<PhoneNumberChooserTask>());

}

public event EventHandler<TaskLaunchEventArgs> TaskLaunchRequested = delegate { };

public void Handle(PhoneNumberResult message)

{

MessageBox.Show("The result was " + message.TaskResult, DisplayName, MessageBoxButton.OK); //You should abstract this…

}

}

Now you should be noticing some interesting tidbits here. I’ll be coming back to those in a minute. Let’s take a look at the last piece in our application’s UI, the view for these tabs:

<UserControl x:Class="Caliburn.Micro.HelloWP7.TabView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<StackPanel Background="{StaticResource PhoneChromeBrush}">

<TextBlock x:Name="DisplayName"

Margin="8 0 0 0"

FontFamily="{StaticResource PhoneFontFamilyNormal}"

FontSize="{StaticResource PhoneFontSizeNormal}"

Foreground="{StaticResource PhoneForegroundBrush}" />

<TextBox x:Name="Text" />

<Button Content="Choose"

x:Name="Choose" />

</StackPanel>

</UserControl>

Here we just have a TextBox bound to our Text property and a Button that will execute our Choose method. Hopefully, you can see how all that comes together in the image above. Let’s add one more thing, just to make our application complete before we dive into the really interesting parts. I mentioned that I had modified my SimpleContainer to create a custom PhoneContainer. All I did was plug in a special Caliburn.Micro piece that helps us use Launchers/Choosers easier (discussed shortly). Pretty much every IoC Container has a place to plug in custom “activation” logic. This is code that gets a chance to execute whenever the container creates an instance of a class. For my simple container, I provided an overridable method to plug this in. It’s a bit different for each container, but it’s a powerful extensibility point. By plugging in CMs InstanceActivator class, we will make launchers/chooser so much easier for you. For now, I’ll show you what the PhoneContainer looks like. We’ll get to the launcher/chooser bit a little later:

using System;

public class PhoneContainer : SimpleContainer

{

public PhoneContainer()

{

Activator = new InstanceActivator(type => GetInstance(type, null));

}

public InstanceActivator Activator { get; private set; }

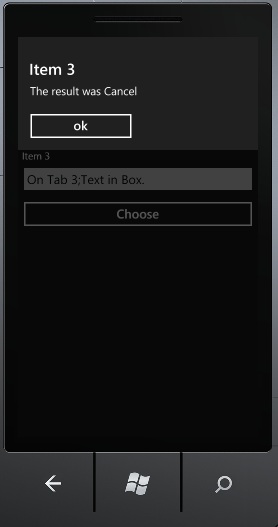
protected override object ActivateInstance(Type type, object[] args)

{

return Activator.ActivateInstance(base.ActivateInstance(type, args));

}

}

**Tombstoning**  
  
Now that we have all the pieces, I can explain why I’ve gone through the trouble of setting all this up: Tombstoning. At any point in time a WP7 application has to expect that it will be shutdown by the OS. For example, if someone is using your application and their friend calls them, the phone will Tombstone your application in order to process the call. When the call is over, your application will be Resurrected. A well designed WP7 application should restore itself to the exact state it was in before Tombstoning. The phone’s OS remembers the page you were on and Navigates to it automatically (read more here). Our Navigation system described above ensures that it gets hooked up with the right ViewModel. But, you must ensure that that View/ViewModel is put back in the same state it was in before Tombstoning. That’s where the SurviveTombstone attribute you saw comes into play. Whenever the OS attempts to Tombstone your application, Caliburn.Micro will grab the current page, extract its DataContext and see if it has any attributes that implement ITombstone. This interface is used to serialize a class or property by flattening it and storing it into the PhoneApplicationService.State property, which is persisted by the OS solely for the purpose of Resurrection (if it occurs). The [SurviveTombstone](http://caliburnmicro.codeplex.com/wikipage?title=SurviveTombstone&referringTitle=Working%20with%20Windows%20Phone%207) attribute is a simple out-of-the-box implementation of this interface that understands how to walk an object graph, examine its properties and persist them. It has special knowledge of Conductors as well, so that it will persist their ActiveItem correctly and inspect their children. This is an opt-in mechanism. You must decorate every class as well as the properties you want persisted. If you look at the classes above, you can see that the PageTwoViewModel is attributed. Because it is a Conductor, the index of its ActiveItem will be persisted and each of its Items will be traversed as well. Because each TabViewModel is attributed, it will be inspected for persistable properties. The Text property is attributed, so that property on each child item will be persisted. This mechanism is customizable and entirely replaceable if you don’t like it. First, you can create your own attributes that implement ITombstone to customize all or part of the process. Second, you can override the SelectInstancesToTombstone and SelectInstancesToResurrect methods on the PhoneBootstrapper to determine what should be persisted to begin with. As I mentioned, by default, we only inspect the DataContext of the current page. But, you could grab several key VMs from your IoC and supply them to the persistence mechanism here if you like. Lastly, you can override the Tombstone and Resurrect methods on the PhoneBootstrapper to replace the built-in mechanism all together. If you download the the sample attached to this article and run the application, you will see that this all works as expected. Try navigating to the second page, selecting a tab other than the first one and typing some text into the text box. Next, click the “Choose” button (we are going to talk about choosers next). After the chooser launches, press the phone’s back button to cancel. Immediately press F5 in Visual Studio to re-attach the debugger (remember that that chooser caused your application to be tombstoned). After the debugger is attached, your application goes back to page two with the proper tab selected and the text filled back in the text box. The chooser result is displayed as well.\*\* Here’s what you should see:  
  
  
  
  
**Launchers and Choosers**  
  
One unique aspect to WP7 development is Tasks. They come in two flavors. Tasks that simply launch a built-in phone application without returning data, such as EmailComposeTask, are called Launchers. Tasks that return data back to your application, such as PhoneNumberChooserTask, are called Choosers. Now, let me tell you how I really feel…the WP7 API for Choosers is a software design abomination. You need to know how it works for your own benefit and to better appreciate what I am going to show you, so let me take a moment to explain. Executing a task is as simple as creating the class and calling its Show method. But, remember that Tombstoning thing? As soon as you call Show, your application gets Tombstoned. So, right now you should be wondering, “But what if I am using a Chooser? How do I get the results back into my application?” Well, the official guidance is that you should have created that Chooser in the constructor of your code-behind file and wired up its completed event. So, when the phone infrastructure tries to resurrect your application, it will navigate to that page, the chooser will be re-created and the event re-wired. When you re-wire to the event, your event handler is called immediately. The infrastructure is trying to fake a cross-app-instance callback. Now, does that sound like it’s going to play nice with MVVM…or any attempt at decent software design? One thing is for sure, you have to be very careful about when you create the Chooser and wire the event. For example, lets say you had the idea that you would create these Choosers as singletons at startup and just wire their events then and there, in order to remove that from your pages. Well, it won’t work. You wired the event too early. It’s also quite possible to wire it too late. It must be done at just the right time. But, what is the right time? From what I can tell, it’s not clearly defined by their API. If you do it in your page’s constructor, it should work though. But should you handle the event in the constructor? Probably not, especially since your main UI isn’t even visible yet. So, you probably at least need to wire for the Loaded event and delay handling of the callback until the UI is visible. As you can see, this part of the WP7 API is a disaster. I’ve done some things to try to improve this and make it more friendly to UI architecture. The following functionality is handled by the InstanceActivator and must be hooked into your IoC container for it to be enabled. Here’s how it works:

* First, developers register the Launcher/Chooser types they intend to use with the Activator. If you look back at the sample’s bootstrapper, you will see that I used the methods InstallLauncher and InstallChooser.
* Each instance that is created by the IoC container will be inspected by the Activator to see if it implements certain interfaces.
* If the instance implements ILaunchTask or ILaunchChooser<TResult>, the activator will wire itself to the event defined by the interface.
* When the VM wants to execute a task, it raises ILaunchTask.TaskLaunchRequested. You can see an example of this in the the TabViewModel.Choose method.
* The infrastructure is notified of the event and then determines if the sender implements IConfigureTask<TTask>. If it does, the task instance is passed to the sender’s Configure method. (Some tasks have parameters which must be set, but the sample above does not.)
* Next, the infrastructure “records” the sender of the event so that it knows who to call back after resurrection.
* The infrastructure now calls Show on the task and your application is Tombstoned.
* When, your application resurrects, it should put itself back in the state it was before tombstoning.
* Since all instances come from the IoC container, we inspect each instance that is created to see if it was the one we “recorded” as the sender of the task launch.
* When we find a match, we attempt to call ILaunchChooser<TResult>.Handle with the Chooser result at the “latest possible time.” The latest possible time is evaluated based on the features of the class. If it is IViewAware, we call handle after the View’s Loaded event fires. If it isn’t but implements IActivate, we call Handle after the class is activated (assuming it isn’t already activated.) If these conditions aren’t met, we call Handle immediately.

It sounds complicated. That’s mostly because, frankly, Choosers are not very friendly to any sort of reasonable software design. However, I think the result of the custom Activator, especially when combined with the [SurviveTombstone](http://caliburnmicro.codeplex.com/wikipage?title=SurviveTombstone&referringTitle=Working%20with%20Windows%20Phone%207) attribute is a very natural developer experience. If you run the sample, you will see that this does indeed work and that the Chooser result is passed back to the correct VM after it’s View is shown.  
  
  
I’ve gone through a lot of WP7 specific stuff in this article. Please remember that all of this is completely optional. You can still leverage Caliburn.Micro without it. I think the FrameAdapter will prove to be the most useful and generalizable feature for building UI. Hopefully the simple tombstoning mechanism and Launcher/Chooser abstraction will help you too.

**Samples**

[Hello WP7](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=140976)  
  
**Footnotes**

1. Even though the IGuardClose interface was designed to handle async scenarios, you must use it synchronously in WP7. This is due to a flaw in the design of Silverlight Navigation Framework.
2. Sometimes the first time I do this, the debugger does not re-attach correctly and the application never resurrects at all. If this happens to you, just try it again ;)

## IResult and Coroutines

Previously, I mentioned that there was one more compelling feature of the Actions concept called Coroutines. If you haven’t heard that term before, here’s what [wikipedia\*](http://en.wikipedia.org/wiki/Coroutine) has to say:  
  
*In computer science, coroutines are program components that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, iterators,infinite lists and pipes.*  
  
Here’s one way you can thing about it: Imagine being able to execute a method, then pause it’s execution on some statement, go do something else, then come back and resume execution where you left off. This technique is extremely powerful in task-based programming, especially when those tasks need to run asynchronously. For example, let’s say we have a ViewModel that needs to call a web service asynchronously, then it needs to take the results of that, do some work on it and call another web service asynchronously. Finally, it must then display the result in a modal dialog and respond to the user’s dialog selection with another asynchronous task. Accomplishing this with the standard event-driven async model is not a pleasant experience. However, this is a simple task to accomplish by using coroutines. The problem…C# doesn’t implement coroutines natively. Fortunately, we can (sort of) build them on top of iterators.  
  
There are two things necessary to take advantage of this feature in Caliburn.Micro: First, implement the IResult interface on some class, representing the task you wish to execute; Second, yield instances of IResult from an Action. Let’s make this more concrete. Say we had a Silverlight application where we wanted to dynamically download and show screens not part of the main package. First we would probably want to show a “Loading” indicator, then asynchronously download the external package, next hide the “Loading” indicator and finally navigate to a particular screen inside the dynamic module. Here’s what the code would look like if your first screen wanted to use coroutines to navigate to a dynamically loaded second screen:

using System.Collections.Generic;

using System.ComponentModel.Composition;

[Export(typeof(ScreenOneViewModel))]

public class ScreenOneViewModel

{

public IEnumerable<IResult> GoForward()

{

yield return Loader.Show("Downloading...");

yield return new LoadCatalog("Caliburn.Micro.Coroutines.External.xap");

yield return Loader.Hide();

yield return new ShowScreen("ExternalScreen");

}

}

First, notice that the Action “GoForward” has a return type of IEnumerable<IResult>. This is critical for using coroutines. The body of the method has four yield statements. Each of these yields is returning an instance of IResult. The first is a result to show the “Downloading” indicator, the second to download the xap asynchronously, the third to hide the “Downloading” message and the fourth to show a new screen from the downloaded xap. After each yield statement, the compiler will “pause” the execution of this method until that particular task completes. The first, third and fourth tasks are synchronous, while the second is asynchronous. But the yield syntax allows you to write all the code in a sequential fashion, preserving the original workflow as a much more readable and declarative structure. To understand a bit more how this works, have a look at the IResult interface:

public interface IResult

{

void Execute(ActionExecutionContext context);

event EventHandler<ResultCompletionEventArgs> Completed;

}

It’s a fairly simple interface to implement. Simply write your code in the “Execute” method and be sure to raise the “Completed” event when you are done, whether it be a synchronous or an asynchronous task. Because coroutines occur inside of an Action, we provide you with an ActionExecutionContext useful in building UI-related IResult implementations. This allows the ViewModel a way to declaratively state it intentions in controlling the view without having any reference to a View or the need for interaction-based unit testing. Here’s what the ActionResultContext looks like:

public class ActionExecutionContext

{

public ActionMessage Message;

public FrameworkElement Source;

public object EventArgs;

public object Target;

public DependencyObject View;

public MethodInfo Method;

public Func<bool> CanExecute;

public object this[string key];

}

And here’s an explanation of what all these properties mean:

* Message – The original ActionMessage that caused the invocation of this IResult.
* Source – The FrameworkElement that triggered the execution of the Action.
* EventArgs – Any event arguments associated with the trigger of the Action.
* Target – The class instance on which the actual Action method exists.
* View – The view associated with the Target.
* Method – The MethodInfo specifying which method to invoke on the Target instance.
* CanExecute – A function that returns true if the Action can be invoked, false otherwise.
* Key Index: A place to store/retrieve any additional metadata which may be used by extensions to the framework.

Bearing that in mind, I wrote a naive Loader IResult that searches the VisualTree looking for the first instance of a BusyIndicator to use to display a loading message. Here’s the implementation:

using System;

using System.Windows;

using System.Windows.Controls;

public class Loader : IResult

{

readonly string message;

readonly bool hide;

public Loader(string message)

{

this.message = message;

}

public Loader(bool hide)

{

this.hide = hide;

}

public void Execute(ActionExecutionContext context)

{

var view = context.View as FrameworkElement;

while(view != null)

{

var busyIndicator = view as BusyIndicator;

if(busyIndicator != null)

{

if(!string.IsNullOrEmpty(message))

busyIndicator.BusyContent = message;

busyIndicator.IsBusy = !hide;

break;

}

view = view.Parent as FrameworkElement;

}

Completed(this, new ResultCompletionEventArgs());

}

public event EventHandler<ResultCompletionEventArgs> Completed = delegate { };

public static IResult Show(string message = null)

{

return new Loader(message);

}

public static IResult Hide()

{

return new Loader(true);

}

}

See how I took advantage of context.View? This opens up a lot of possibilities while maintaining separation between the view and the view model. Just to list a few interesting things you could do with IResult implementations: show a message box, show a VM-based modal dialog, show a VM-based Popup at the user’s mouse position, play an animation, show File Save/Load dialogs, place focus on a particular UI element based on VM properties rather than controls, etc. Of coarse, one of the biggest opportunities is calling web services. Let’s look at how you might do that, but by using a slightly different scenario, dynamically downloading a xap:

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

using System.ComponentModel.Composition.ReflectionModel;

using System.Linq;

public class LoadCatalog : IResult

{

static readonly Dictionary<string, DeploymentCatalog> Catalogs = new Dictionary<string, DeploymentCatalog>();

readonly string uri;

[Import]

public AggregateCatalog Catalog { get; set; }

public LoadCatalog(string relativeUri)

{

uri = relativeUri;

}

public void Execute(ActionExecutionContext context)

{

DeploymentCatalog catalog;

if(Catalogs.TryGetValue(uri, out catalog))

Completed(this, new ResultCompletionEventArgs());

else

{

catalog = new DeploymentCatalog(uri);

catalog.DownloadCompleted += (s, e) =>{

if(e.Error == null)

{

Catalogs[uri] = catalog;

Catalog.Catalogs.Add(catalog);

catalog.Parts

.Select(part => ReflectionModelServices.GetPartType(part).Value.Assembly)

.Where(assembly => !AssemblySource.Instance.Contains(assembly))

.Apply(x => AssemblySource.Instance.Add(x));

}

else Loader.Hide().Execute(context);

Completed(this, new ResultCompletionEventArgs {

Error = e.Error,

WasCancelled = false

});

};

catalog.DownloadAsync();

}

}

public event EventHandler<ResultCompletionEventArgs> Completed = delegate { };

}

In case it wasn’t clear, this sample is using MEF. Furthermore, we are taking advantage of the DeploymentCatalog created for Silverlight 4. You don’t really need to know a lot about MEF or DeploymentCatalog to get the takeaway. Just take note of the fact that we wire for the DownloadCompleted event and make sure to fire the IResult.Completed event in its handler. This is what enables the async pattern to work. We also make sure to check the error and pass that along in the ResultCompletionEventArgs. Speaking of that, here’s what that class looks like:

public class ResultCompletionEventArgs : EventArgs

{

public Exception Error;

public bool WasCancelled;

}

Caliburn.Micro’s enumerator checks these properties after it get’s called back from each IResult. If there is either an error or WasCancelled is set to true, we stop execution. You can use this to your advantage. Let’s say you create an IResult for the OpenFileDialog. You could check the result of that dialog, and if the user canceled it, set WasCancelled on the event args. By doing this, you can write an action that assumes that if the code following the Dialog.Show executes, the user must have selected a file. This sort of technique can simplify the logic in such situations. Obviously, you could use the same technique for the SaveFileDialog or any confirmation style message box if you so desired. My favorite part of the LoadCatalog implementation shown above, is that the original implementation was written by a CM user! Thanks janoveh for this awesome submission! As a side note, one of the things we added to the CM project site is a “Recipes” section. We are going to be adding more common solutions such as this to that area in the coming months. So, it will be a great place to check for cool plugins and customizations to the framework.  
  
Another thing you can do is create a series of IResult implementations built around your application’s shell. That is what the ShowScreen result used above does. Here is its implementation:

using System;

using System.ComponentModel.Composition;

public class ShowScreen : IResult

{

readonly Type screenType;

readonly string name;

[Import]

public IShell Shell { get; set; }

public ShowScreen(string name)

{

this.name = name;

}

public ShowScreen(Type screenType)

{

this.screenType = screenType;

}

public void Execute(ActionExecutionContext context)

{

var screen = !string.IsNullOrEmpty(name)

? IoC.Get<object>(name)

: IoC.GetInstance(screenType, null);

Shell.ActivateItem(screen);

Completed(this, new ResultCompletionEventArgs());

}

public event EventHandler<ResultCompletionEventArgs> Completed = delegate { };

public static ShowScreen Of<T>()

{

return new ShowScreen(typeof(T));

}

}

This bring up another important feature of IResult. Before CM executes a result, it passes it through the IoC.BuildUp method allowing your container the opportunity to push dependencies in through the properties. This allows you to create them normally within your view models, while still allowing them to take dependencies on application services. In this case, we depend on IShell. You could also have your container injected, but in this case I chose to use the IoC static class internally. As a general rule, you should avoid pulling things from the container directly. However, I think it is acceptable when done inside of infrastructure code such as a ShowScreen IResult.  
  
I hope this gives some explanation and creative ideas for what can be accomplished with IResult. Be sure to check out the sample application attached. There’s a few other interesting things in there as well.

### Samples

[Coroutines](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=144749)  
  
**Footnotes**

1. When I went to look up the “official” definition on wikipedia I was interested to see what it had to say about implementations in various languages. Scrolling down to the section on C#…Caliburn was listed! Fun stuff.
2. You can also return a single instance of IResult without using IEnuermable<IResult> if you just want a simple way to execute a single task.

## Screens, Conductors and Composition

Actions, Coroutines and Conventions tend to draw the most attention to Caliburn.Micro, but the Screens and Conductors piece is probably most important to understand if you want your UI to be engineered well. It’s particularly important if you want to leverage composition. The terms Screen, Screen Conductor and Screen Collection have more recently been codified by Jeremy Miller during his work on the book "Presentation Patterns" for Addison Wesley. While these patterns are primarily used in CM by inheriting ViewModels from particular base classes, its important to think of them as roles rather than as View-Models. In fact, depending on your architecture, a Screen could a be a UserControl, Presenter or ViewModel. That’s getting a little ahead of ourselves though. First, let’s talk about what these things are in general.

### Theory

**Screen**  
  
This is the simplest construct to understand. You might think of it as a stateful unit of work existing within the presentation tier of an application. It’s independent from the application shell. The shell may display many different screens, some even at the same time. The shell may display lots of widgets as well, but these are not part of any screen. Some screen examples might be a modal dialog for application settings, a code editor window in Visual Studio or a page in a browser. You probably have a pretty good intuitive sense about this.  
  
Often times a screen has a lifecycle associated with it which allows the screen to perform custom activation and deactivation logic. This is what Jeremy calls the ScreenActivator. For example, take the Visual Studio code editor window. If you are editing a C# code file in one tab, then you switch to a tab containing an XML document, you will notice that the toolbar icons change. Each one of those screens has custom activation/deactivation logic that enables it to setup/teardown the application toolbars such that they provide the appropriate icons based on the active screen. In simple scenarios, the ScreenActivator is often the same class as the Screen. However, you should remember that these are two separate roles. If a particular screen has complex activation logic, it may be necessary to factor the ScreenActivator into its own class in order to reduce the complexity of the Screen. This is particularly important if you have an application with many different screens, but all with the same activation/deactivation logic.  
  
**Screen Conductor**  
  
Once you introduce the notion of a Screen Activation Lifecycle into your application, you need some way to enforce it. This is the role of the ScreenConductor. When you show a screen, the conductor makes sure it is properly activated. If you are transitioning away from a screen, it makes sure it gets deactivated. There’s another scenario that’s important as well. Suppose that you have a screen which contains unsaved data and someone tries to close that screen or even the application. The ScreenConductor, which is already enforcing deactivation, can help out by implementing Graceful Shutdown. In the same way that your Screen might implement an interface for activation/deactivation, it may also implement some interface which allows the conductor to ask it “Can you close?” This brings up an important point: in some scenarios deactivating a screen is the same as closing a screen and in others, it is different. For example, in Visual Studio, it doesn’t close documents when you switch from tab to tab. It just activates/deactivates them. You have to explicitly close the tab. That is what triggers the graceful shutdown logic. However, in a navigation based application, navigating away from a page would definitely cause deactivation, but it might also cause that page to close. It all depends on your specific application’s architecture and it’s something you should think carefully about.  
  
**Screen Collection**  
  
In an application like Visual Studio, you would not only have a ScreenConductor managing activation, deactivation, etc., but would also have a ScreenCollection maintaining the list of currently opened screens or documents. By adding this piece of the puzzle, we can also solve the issue of deactivation vs. close. Anything that is in the ScreenCollection remains open, but only one of those items is active at a time. In an MDI-style application like VS, the conductor would manage switching the active screen between members of the ScreenCollection. Opening a new document would add it to the ScreenCollection and switch it to the active screen. Closing a document would not only deactivate it, but would remove it from the ScreenCollection. All that would be dependent on whether or not it answers the question “Can you close?” positively. Of course, after the document is closed, the conductor needs to decide which of the other items in the ScreenCollection should become the next active document.

### Implementations

There are lots of different ways to implement these ideas. You could inherit from a TabControl and implement an IScreenConductor interface and build all the logic directly in the control. Add that to your IoC container and you’re off and running. You could implement an IScreen interface on a custom UserControl or you could implement it as a POCO used as a base for Supervising Controllers. ScreenCollection could be a custom collection with special logic for maintaining the active screen, or it could just be a simple IList<IScreen>.  
  
Caliburn.Micro Implementations  
  
These concepts are implemented in CM through various interfaces and base classes which can be used mostly(1) to build ViewModels. Let’s take a look at them:  
  
**Screens**  
  
In Caliburn.Micro we have broken down the notion of screen activation into several interfaces:

* IActivate – Indicates that the implementer requires activation. This interface provides an Activate method, an IsActive property and an Activated event which should be raised when activation occurs.
* IDeactivate – Indicates that the implementer requires deactivation. This interface has a Deactivate method which takes a bool property indicating whether to close the screen in addition to deactivating it. It also has two events: AttemptingDeactivation, which should be raised before deactivation and Deactivated which should be raised after deactivation.
* IGuardClose – Indicates that the implementer may need to cancel a close operation. It has one method: CanClose. This method is designed with an async pattern, allowing complex logic such as async user interaction to take place while making the close decision. The caller will pass an Action<bool> to the CanClose method. The implementer should call the action when guard logic is complete. Pass true to indicate that the implementer can close, false otherwise.

In addition to these core lifecycle interfaces, we have a few others to help in creating consistency between presentation layer classes:

* IHaveDisplayName – Has a single property called DisplayName
* INotifyPropertyChangedEx – This interface inherits from the standard INotifyPropertyChanged and augments it with additional behaviors. It adds an IsNotifying property which can be used to turn off/on all change notification, a NotifyOfPropertyChange method which can be called to raise a property change and a Refresh method which can be used to refresh all bindings on the object.
* IObservableCollection<T> – Composes the following interfaces: IList<T>, INotifyPropertyChangedEx and INotifyCollectionChanged
* IChild<T> – Implemented by elements that are part of a hierarchy or that need a reference to an owner. It has one property named Parent.
* IViewAware – Implemented by classes which need to be made aware of the view that they are bound to. It has an AttachView method which is called by the framework when it binds the view to the instance. It has a GetView method which the framework calls before creating a view for the instance. This enables caching of complex views or even complex view resolution logic. Finally, it has an event which should be raised when a view is attached to the instance called ViewAttached.

Because certain combinations are so common, we have some convenience interfaces and base classes:

* PropertyChangedBase – Implements INotifyPropertyChangedEx (and thus INotifyPropertyChanged). It provides a lambda-based NotifyOfPropertyChange method in addition to the standard string mechanism, enabling strongly-typed change notification. Also, all property change events are automatically marshaled to the UI thread.(2)
* BindableCollection – Implements IObservableCollection<T> by inheriting from the standard ObservableCollection<T> and adding the additional behavior specified by INotifyPropertyChangedEx. Also, this class ensures that all property change and collection change events occur on the UI thread.(2)
* IScreen – This interface composes several other interfaces: IHaveDisplayName, IActivate, IDeactivate, IGuardClose and INotifyPropertyChangedEx
* Screen – Inherits from PropertyChangedBase and implements the IScreen interface. Additionally, IChild<IConductor> and IViewAware are implemented.

What this all means is that you will probably inherit most of your view models from either PropertyChangedBase or Screen. Generally speaking, you would use Screen if you need any of the activation features and PropertyChangedBase for everything else. CM’s default Screen implementation has a few additional features as well and makes it easy to hook into the appropriate parts of the lifecycle:

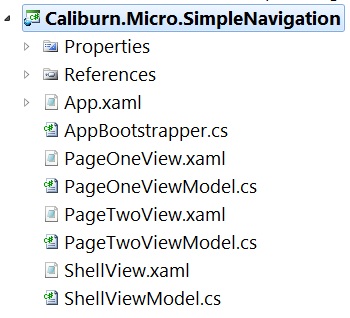
* OnInitialize – Override this method to add logic which should execute only the first time that the screen is activated. After initialization is complete, IsInitialized will be true.
* OnActivate – Override this method to add logic which should execute every time the screen is activated. After activation is complete, IsActive will be true.
* OnDeactivate – Override this method to add custom logic which should be executed whenever the screen is deactivated or closed. The bool property will indicated if the deactivation is actually a close. After deactivation is complete, IsActive will be false.
* CanClose – The default implementation always allows closing. Override this method to add custom guard logic.
* OnViewLoaded – Since Screen implements IViewAware, it takes this as an opportunity to let you know when your view’s Loaded event is fired. Use this if you are following a SupervisingController or PassiveView style and you need to work with the view. This is also a place to put view model logic which may be dependent on the presence of a view even though you may not be working with the view directly.
* TryClose – Call this method to close the screen. If the screen is being controlled by a Conductor, it asks the Conductor to initiate the shutdown process for the Screen. If the Screen is not controlled by a Conductor, but exists independently (perhaps because it was shown using the WindowManager), this method attempts to close the view. In both scenarios the CanClose logic will be invoked and if allowed, OnDeactivate will be called with a value of true.

So, just to re-iterate: if you need a lifecycle, inherit from Screen; otherwise inherit from PropertyChangedBase.  
  
**Conductors**  
  
As I mentioned above, once you introduce lifecycle, you need something to enforce it. In Caliburn.Micro, this role is represented by the IConductor interface which has the following members:

* ActiveItem – A property that indicates what item the conductor is currently tracking as active.
* GetConductedItems – Call this method to return a list of all the items that the conductor is tracking. If the conductor is using a “screen collection,” this returns all the “screens,” otherwise this only returns ActiveItem.
* ActivateItem – Call this method to activate a particular item. It will also add it to the currently conducted items if the conductor is using a “screen collection.”
* CloseItem – Call this method to close a particular item. It will also remove it from the currently conducted items if the conductor is using a “screen collection.”
* ActivationProcessed – Raised when the conductor has processed the activation of an item. It indicates whether or not the activation was successful.(3)
* INotifyPropertyChangedEx – This interface is composed into IConductor.

You may have noticed that CM’s IConductor interface uses the term “item” rather than “screen” and that I was putting the term “screen collection” in quotes. The reason for this is that CM’s conductor implementations do not require the item being conducted to implement IScreen or any particular interface. Conducted items can be POCOs. Rather than enforcing the use of IScreen, each of the conductor implementations is generic, with no constraints on the type. As the conductor is asked to activate/deactivate/close/etc each of the items it is conducting, it checks them individually for the following fine-grained interfaces: IActivate, IDeactive, IGuardClose and IChild<IConductor>. In practice, I usually inherit conducted items from Screen, but this gives you the flexibility to use your own base class, or to only implement the interfaces for the lifecycle events you care about on a per-class basis. You can even have a conductor tracking heterogeneous items, some of which inherit from Screen and others that implement specific interfaces or none at all.  
  
Out of the box CM has two implementations of IConductor, one that works with a “screen collection” and one that does not. We’ll look at the conductor without the collection first.   
  
*Conductor<T>*  
  
This simple conductor implements the majority of IConductor’s members through explicit interface mechanisms and adds strongly typed versions of the same methods which are available publicly. This allows working with conductors generically through the interface as well as in a strongly typed fashion based on the items they are conducting. The Conductor<T> treats deactivation and closing synonymously. Since the Conductor<T> does not maintain a “screen collection,” the activation of each new item causes both the deactivation and close of the previously active item. The actual logic for determining whether or not the conducted item can close can be complex due to the async nature of IGuardClose and the fact that the conducted item may or may not implement this interface. Therefore, the conductor delegates this to an ICloseStrategy<T> which handles this and tells the conductor the results of the inquiry. Most of the time you’ll be fine with the DefaultCloseStrategy<T> that is provided automatically, but should you need to change things (perhaps IGuardClose is not sufficient for your purposes) you can set the CloseStrategy property on Conductor<T> to your own custom strategy.  
  
*Conductor<T>.Collection.OneActive*  
  
This implementation has all the features of Conductor<T> but also adds the notion of a “screen collection.” Since conductors in CM can conduct any type of class, this collection is exposed through an IObservableCollection<T> called Items rather than Screens. As a result of the presence of the Items collection, deactivation and closing of conducted items are not treated synonymously. When a new item is activated, the previous active item is deactivated only and it remains in the Items collection. To close an item with this conductor, you must explicitly call its CloseItem method.(4) When an item is closed and that item was the active item, the conductor must then determine which item should be activated next. Be default, this is the item before the previous active item in the list. If you need to change this behavior, you can override DetermineNextItemToActivate.  
  
There are two very important details about both of CMs IConductor implementations that I have not mentioned yet. First, they both inherit from Screen. This is a key feature of these implementations because it creates a composite pattern between screens and conductors. So, let’s say you are building a basic navigation-style application. Your shell would be an instance of Conductor<IScreen> because it shows one Screen at a time and doesn’t maintain a collection. But, let’s say that one of those screens was very complicated and needed to have a multi-tabbed interface requiring lifecycle events for each tab. Well, that particular screen could inherit from Conductor<T>.Collection.OneActive. The shell need not be concerned with the complexity of the individual screen. One of those screens could even be a UserControl that implemented IScreen instead of a ViewModel if that’s what was required. The second important detail is a consequence of the first. Since all OOTB implementations of IConductor inherit from Screen it means that they too have a lifecycle and that lifecycle cascades to whatever items they are conducting. So, if a conductor is deactivated, it’s ActiveItem will be deactivated as well. If you try to close a conductor, it’s going to only be able to close if all of the items it conducts can close. This turns out to be a very powerful feature. There’s one aspect about this that I’ve noticed frequently trips up developers. If you activate an item in a conductor that is itself not active, that item won’t actually be activated until the conductor gets activated. This makes sense when you think about it, but can occasionally cause hair pulling.  
  
**Quasi-Conductors**  
  
Not everything in CM that can be a screen is rooted inside of a Conductor. For example, what about the your root view model? If it’s a conductor, who is activating it? Well, that’s one of the jobs that the Bootstrapper performs. The Bootstrapper itself is not a conductor, but it understands the fine-grained lifecycle interfaces discussed above and ensures that your root view model is treated with the respect it deserves. The WindowManager works in a similar way(5) by acting a little like a conductor for the purpose of enforcing the lifecycle of your modal (and modeless - WPF only) windows. So, lifecycle isn’t magical. All your screens/conductors must be either rooted in a conductor or managed by the Bootstrapper or WindowManager to work properly; otherwise you are going to need to manage the lifecycle yourself.  
  
*View-First*  
  
If you are working with WP7 or using the Silverlight Navigation Framework, you may be wondering if/how you can leverage screens and conductors. So far, I’ve been assuming a primarily ViewModel-First approach to shell engineering. But the WP7 platform enforces a View-First approach by controlling page navigation. The same is true of the SL Nav framework. In these cases, the Phone/Nav Framework acts like a conductor. In order to make this play well with ViewModels, the WP7 version of CM has a FrameAdapter which hooks into the NavigationService. This adapter, which is set up by the PhoneBootstrapper, understands the same fine-grained lifecycle interfaces that conductors do and ensures that they are called on your ViewModels at the appropriate points during navigation. You can even cancel the phone’s page navigation by implementing IGuardClose on your ViewModel. While the FrameAdapter is only part of the WP7 version of CM, it should be easily portable to Silverlight should you wish to use it in conjunction with the Silverlight Navigation Framework.

### Simple Navigation

  
Previously, we discussed the theory and basic APIs for Screens and Conductors in Caliburn.Micro. Now I would like to walk through the first of several samples. This particular sample demonstrates how to set up a simple navigation-style shell using Conductor<T> and two “Page” view models. As you can see from the project structure, we have the typical pattern of Bootstrapper and ShellViewModel. In order to keep this sample as simple as possible, I’m not even using an IoC container with the Bootstrapper. Let’s look at the ShellViewModel first. It inherits from Conductor<object> and is implemented as follows:

public class ShellViewModel : Conductor<object> {

public ShellViewModel() {

ShowPageOne();

}

public void ShowPageOne() {

ActivateItem(new PageOneViewModel());

}

public void ShowPageTwo() {

ActivateItem(new PageTwoViewModel());

}

}

Here is the corresponding ShellView:

<UserControl x:Class="Caliburn.Micro.SimpleNavigation.ShellView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:tc="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit">

<tc:DockPanel>

<StackPanel Orientation="Horizontal"

HorizontalAlignment="Center"

tc:DockPanel.Dock="Top">

<Button x:Name="ShowPageOne"

Content="Show Page One" />

<Button x:Name="ShowPageTwo"

Content="Show Page Two" />

</StackPanel>

<ContentControl x:Name="ActiveItem" />

</tc:DockPanel>

</UserControl>

Notice that the ShellViewModel has two methods, each of which passes a view model instance to the ActivateItem method. Recall from our earlier discussion that ActivateItem is a method on Conductor<T> which will switch the ActiveItem property of the conductor to this instance and push the instance through the activation stage of the screen lifecycle (if it supports it by implementing IActivate). Recall also, that if ActiveItem is already set to an instance, then before the new instance is set, the previous instance will be checked for an implementation of IGuardClose which may or may not cancel switching of the ActiveItem. Assuming the current ActiveItem can close, the conductor will then push it through the deactivation stage of the lifecycle, passing true to the Deactivate method to indicate that the view model should also be closed. This is all it takes to create a navigation application in Caliburn.Micro. The ActiveItem of the conductor represents the “current page” and the conductor manages the transitions from one page to the other. This is all done in a ViewModel-First fashion since its the conductor and it’s child view models that are driving the navigation and not the “views.”  
  
Once the basic Conductor structure is in place, it’s quite easy to get it rendering. The ShellView demonstrates this. All we have to do is place a ContentControl in the view. By naming it “ActiveItem” our data binding conventions kick in. The convention for ContentControl is a bit interesting. If the item we are binding to is not a value type and not a string, then we assume that the Content is a ViewModel. So, instead of binding to the Content property as we would in the other cases, we actually set up a binding with CM’s custom attached property: View.Model. This property causes CM’s ViewLocator to look up the appropriate view for the view model and CM’s ViewModelBinder to bind the two together. Once that is complete, we pop the view into the ContentControl’s Content property. This single convention is what enables the powerful, yet simple ViewModel-First composition in the framework.  
  
For completeness, let’s take a look at the PageOneViewModel and PageTwoViewModel:

public class PageOneViewModel {}

public class PageTwoViewModel : Screen {

protected override void OnActivate() {

MessageBox.Show("Page Two Activated"); //Don't do this in a real VM.

base.OnActivate();

}

}

Along with their views:

<UserControl x:Class="Caliburn.Micro.SimpleNavigation.PageOneView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<TextBlock FontSize="32">Page One</TextBlock>

</UserControl>

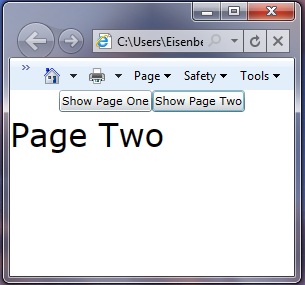
<UserControl x:Class="Caliburn.Micro.SimpleNavigation.PageTwoView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<TextBlock FontSize="32">Page Two</TextBlock>

</UserControl>

I’ve intentionally kept this bare bones so that it’s easy to play around with and see how these pieces work together. It’s not the slightest bit impressive, but here’s what it looks like:  
  
  
  
I’d like to point out a few final things. Notice that PageOneViewModel is just a POCO, but PageTwoViewModel inherits from Screen. Remember that the conductors in CM don’t place any constraints on what can be conducted. Rather, they check each instance for support of the various fine-grained lifecycle instances at the necessary times. So, when ActivateItem is called for PageTwoViewModel, it will first check PageOneViewModel to see if it implements IGuardClose. Since it does not, it will attempt to close it. It will then check to see if it implements IDeactivate. Since it does not, it will just proceed to activate the new item. First it checks if the new item implements IChild<IConductor>. Since Screen does, it hooks up the hierarchical relationship. Next, it will check PageTwoViewModel to see if it implements IActivate. Since Screen does, the code in my OnActivate method will then run. Finally, it will set the ActiveItem property on the conductor and raise the appropriate events. Here’s an important consequence of this that should be remembered: The activation is a ViewModel-specific lifecycle process and doesn’t guarantee anything about the state of the View. Many times, even though your ViewModel has been activated, it’s view may not yet be visible. You will see this when you run the sample. The MessageBox will show when the activation occurs, but the view for page two will not yet be visible. Remember, if you have any activation logic that is dependent on the view being already loaded, you should override Screen.OnViewLoaded instead of/in combination with OnActivate.

### Samples

[Simple Navigation](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=156876)  
  
**Footnotes**  
  
1. These classes can also be used very easily to create a SupervisingController or even a PassiveView design if so desired. Normally, using these classes as view models will suffice, but since the base implementations Screen, Conductor<T> and Conductor<T>.Collection.OneActive all implement IViewAware, it’s easy to get a reference to the view and take one of these alternative approaches. Additionally, since all Caliburn.Micro code depends only on the interfaces, you could easily implement IConductor on top of a docking window manager or some other complex control or service.  
  
2. Even though these classes do automatic UI thread marshalling, they are still safe to use in a unit test. Under the covers, these classes use CM’s Execute.OnUIThread utility method. This method calls a custom action to do the thread marshalling which is setup by the Bootstrapper. If you are working with these classes without running the Bootstrapper (as would be the case with testing), a default action is used which does not do any marshalling.  
  
3. If activation would cause the closing of the ActiveItem, such as it would with Conductor<T>, and the active item’s CanClose returned false, then activation of the new item would not succeed. The current item would remain active.  
  
4. Or if the item inherits from Screen, you can call the Screen’s TryClose method, which actually just calls Parent.CloseItem passing itself as the item to be closed.  
  
5. In fact the WPF version of the Bootstrapper uses the WindowManager internally to show your MainWindow. I’ll have a whole article on the WindowManager coming soon.

# Recipes

## SimpleContainer

Caliburn.Micro is best used with an Inversion of Control (IoC) Container. Because many developers have a preference for a particular container, such a component is not included out-of-the-box with CM. Should you find yourself in a situation where you are building a small application or something for WP7 where you cannot or do not want to use a full featured IoC container, the following code should provide you with a solution.  
  
**Features**

* Singleton Registration
* PerRequest (Transient) Registration
* Instance Registration
* Registration of Arbitrary Handlers
* Get a single instance by Type and/or key.
* Get all instances of a particular type.
* Automatic Func<T> factory creation.
* Automatic greedy constructor injection.
* Constructor injection of IEnumerable<TService>
* Overridable instance activation.

**Implementation**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

public class SimpleContainer

{

readonly List<ContainerEntry> entries = new List<ContainerEntry>();

public void RegisterInstance(Type service, string key, object implementation)

{

RegisterHandler(service, key, () => implementation);

}

public void RegisterPerRequest(Type service, string key, Type implementation)

{

RegisterHandler(service, key, () => BuildInstance(implementation));

}

public void RegisterSingleton(Type service, string key, Type implementation)

{

object singleton = null;

RegisterHandler(service, key, () => singleton ?? (singleton = BuildInstance(implementation)));

}

public void RegisterHandler(Type service, string key, Func<object> handler)

{

GetOrCreateEntry(service, key).Add(handler);

}

public object GetInstance(Type service, string key)

{

var entry = GetEntry(service, key);

if(entry != null)

return entry.Single()();

if(typeof(Delegate).IsAssignableFrom(service))

{

var typeToCreate = service.GetGenericArguments()[0];

var factoryFactoryType = typeof(FactoryFactory<>).MakeGenericType(typeToCreate);

var factoryFactoryHost = Activator.CreateInstance(factoryFactoryType);

var factoryFactoryMethod = factoryFactoryType.GetMethod("Create");

return factoryFactoryMethod.Invoke(factoryFactoryHost, new object[] { this });

}

else if(typeof(IEnumerable).IsAssignableFrom(service))

{

var listType = service.GetGenericArguments()[0];

var instances = GetAllInstances(listType).ToList();

var array = Array.CreateInstance(listType, instances.Count);

for(var i = 0; i < array.Length; i++)

{

array.SetValue(instances[i], i);

}

return array;

}

return null;

}

public IEnumerable<object> GetAllInstances(Type service)

{

var entry = GetEntry(service, null);

return entry != null ? entry.Select(x => x()) : new object[0];

}

public void BuildUp(object instance)

{

var injectables = from property in instance.GetType().GetProperties()

where property.CanRead && property.CanWrite && property.PropertyType.IsInterface

select property;

injectables.Apply(x =>{

var injection = GetAllInstances(x.PropertyType);

if(injection.Any())

x.SetValue(instance, injection.First(), null);

});

}

ContainerEntry GetOrCreateEntry(Type service, string key)

{

var entry = GetEntry(service, key);

if (entry == null)

{

entry = new ContainerEntry { Service = service, Key = key };

entries.Add(entry);

}

return entry;

}

ContainerEntry GetEntry(Type service, string key)

{

return service == null

? entries.Where(x => x.Key == key).FirstOrDefault()

: entries.Where(x => x.Service == service && x.Key == key).FirstOrDefault();

}

protected object BuildInstance(Type type)

{

var args = DetermineConstructorArgs(type);

return ActivateInstance(type, args);

}

protected virtual object ActivateInstance(Type type, object[] args)

{

return args.Length > 0 ? Activator.CreateInstance(type, args) : Activator.CreateInstance(type);

}

object[] DetermineConstructorArgs(Type implementation)

{

var args = new List<object>();

var constructor = SelectEligibleConstructor(implementation);

if (constructor != null)

args.AddRange(constructor.GetParameters().Select(info => GetInstance(info.ParameterType, null)));

return args.ToArray();

}

static ConstructorInfo SelectEligibleConstructor(Type type)

{

return (from c in type.GetConstructors()

orderby c.GetParameters().Length descending

select c).FirstOrDefault();

}

class ContainerEntry : List<Func<object>>

{

public string Key;

public Type Service;

}

class FactoryFactory<T>

{

public Func<T> Create(SimpleContainer container)

{

return () => (T)container.GetInstance(typeof(T), null);

}

}

}

## LoadCatalog Result

The following is a simple IResult implementation for using MEF's DeploymentCatalog to dynamically download and plug in xaps. In order for it to work properly, you must make sure to add the AggregateCatalog as an exported value in the container.  
  
**Implementation**

using System;

using System.Collections.Generic;

using System.ComponentModel.Composition;

using System.ComponentModel.Composition.Hosting;

using System.ComponentModel.Composition.ReflectionModel;

using System.Linq;

public class LoadCatalog : IResult

{

static readonly Dictionary<string, DeploymentCatalog> Catalogs = new Dictionary<string, DeploymentCatalog>();

readonly string uri;

[Import]

public AggregateCatalog Catalog { get; set; }

public LoadCatalog(string relativeUri)

{

uri = relativeUri;

}

public void Execute(ActionExecutionContext context)

{

DeploymentCatalog catalog;

if(Catalogs.TryGetValue(uri, out catalog))

Completed(this, new ResultCompletionEventArgs());

else

{

catalog = new DeploymentCatalog(uri);

catalog.DownloadCompleted += (s, e) =>{

if(e.Error == null)

{

Catalogs[uri] = catalog;

Catalog.Catalogs.Add(catalog);

catalog.Parts

.Select(part => ReflectionModelServices.GetPartType(part).Value.Assembly)

.Where(assembly => !AssemblySource.Instance.Contains(assembly))

.Apply(x => AssemblySource.Instance.Add(x));

}

Completed(this, new ResultCompletionEventArgs {

Error = e.Error,

WasCancelled = false

});

};

catalog.DownloadAsync();

}

}

public event EventHandler<ResultCompletionEventArgs> Completed = delegate { };

}

**Usage**

public IEnumerable<IResult> LoadCatalogs()

{

yield return new LoadCatalog("TestCatalog1.xap");

yield return new LoadCatalog("TestCatalog2.xap");

yield return new LoadCatalog("TestCatalog3.xap");

yield return new LoadCatalog("TestCatalog4.xap");

}

Much thanks to codeplex user janoveh on whose work this recipe was based.

## Testing Coroutines

Here's a few simple steps to test a method that utilizes coroutines:

1. When testing a coroutine, simply enumerate each result, one at a time. Do not call execute on your results. Remember, you are not testing the IResult implementation, but the method which uses it.
2. Check to make sure that the correct IResult is yielded and check any properties it has which would indicate it's correct/incorrect configuration.
3. Set any properties on the IResult which would have been set as a result of execution....but do not call execute.
4. Go to beginning.

By following these steps you will be able to "ignore" the async part of the coroutine (if it has one) by exerting direct control over the enumeration. You can even choose to stop enumerating at any point once you have confirmed that your test passes/fails. You will probably want to use several test methods to test a single coroutine implementation in order to keep individual tests simple and small.  
  
Here's a sample class you might use to help in enumerating the results in tests:

public class CoroutineEnumerator

{

private readonly IEnumerator<IResult> enumerator;

public CoroutineEnumerator(IEnumerable<IResult> enumerable)

{

this.enumerator = enumerable.GetEnumerator();

}

public CoroutineEnumerator(IEnumerator<IResult> enumerator)

{

this.enumerator = enumerator;

}

public TCoroutine Next<TCoroutine>()

{

while(enumerator.MoveNext())

{

if(enumerator.Current is TCoroutine)

return (TCoroutine)enumerator.Current;

}

throw new InvalidOperationException("List of coroutines does not include " + typeof(TCoroutine).Name);

}

public void Finish()

{

while(enumerator.MoveNext()) {}

}

}

## Action Filters

One of the features missing in Caliburn.Micro is filters, a set of action decorations aimed to provide additional behaviors to a particular action. Filters bring two major advantages:

* They help to keep View Model free from noisy code that is unrelated to its main purpose. This greatly simplifies unit testing.
* They allow the developer to share the implementation of cross-cutting concerns between different View Models by moving it into infrastructure components.

[Download the Sample Solution](http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=caliburnmicro&DownloadId=150463)

## Design

Filters falls into two main categories, depending on the mechanism used to interact with the target action: IExecutionWrapper and IContextAware. The common IFilter interface is little more than a marker and just defines a Priority property used to guide the filter application order:

public interface IFilter

{

int Priority { get; }

}

### IExecutionWrapper

Most filter frameworks are built around AOP concepts and work through the interception of action execution. Doing this allows you to add operations before and after the action execution itself or something more complex like dispatching the action in another thread. The coroutine infrastructure already has this capability, so a filter willing to intercept an action can simply wrap the original execution into a “wrapping” IResult:

public interface IExecutionWrapper : IFilter

{

IResult Wrap(IResult inner);

}

To enable filter hooking, replace the core invocation method of ActionMessage:

//in bootstrapper code

...

FilterFramework.Configure();

...

public static class FilterFramework

{

public static void Configure()

{

...

ActionMessage.InvokeAction = InvokeAction;

}

...

public static void InvokeAction(ActionExecutionContext context)

{

var values = MessageBinder.DetermineParameters(context, context.Method.GetParameters());

var result = Coroutine.CreateParentEnumerator(ExecuteActionWithParameters(values).GetEnumerator());

var wrappers = FilterManager.GetFiltersFor(context).OfType<IExecutionWrapper>();

var pipeline = result.WrapWith(wrappers);

//if pipeline has error, action execution should throw!

pipeline.Completed += (o, e) => {

Execute.OnUIThread(() => {

if(e.Error != null) {

throw new Exception(

string.Format("An error occurred while executing {0}.", context.Message),

e.Error

);

}

});

};

pipeline.Execute(context);

}

...

}

Every action is actually executed within an ExecuteActionResult (code omitted here) that deals with simple actions as well as coroutines. It does this by conforming all of them to a common IResult interface. This “core” IResult is afterwards wrapped over and over by each filter attached to the action and finally executed.  
  
Let’s have a look at FilterManager:

public static class FilterManager {

public static Func<ActionExecutionContext, IEnumerable<IFilter>> GetFiltersFor = context => {

//TODO: apply caching?

return context.Target.GetType().GetAttributes<IFilter>(true)

.Union(context.Method.GetAttributes<IFilter>(true))

.OrderBy(x => x.Priority);

};

public static IResult WrapWith(this IResult inner, IEnumerable<IExecutionWrapper> wrappers) {

return wrappers.Aggregate(inner, (current, wrapper) => wrapper.Wrap(current));

}

}

Note that the GetFiltersFor method is replaceable to allow for another filter lookup strategy (for example, based on convention or external configuration instead of attributes).

### IContextAware

While IExecutionWrapper-s do their work during the action execution, the other filter category, IContextAware, operates when an action is not executing, providing preconditions for execution (the related predicate is held by ActionExecutionContext) or observing the ViewModel to force an update of the action availability:

public interface IContextAware : IFilter, IDisposable

{

void MakeAwareOf(ActionExecutionContext context);

}

Filters implementing this interface are given a chance, during ActionMessage initialization, to hook the execution context. To achieve this, the framework hooks into ActionMessage.PrepareContext:

public static class FilterFramework {

public static void Configure() {

var oldPrepareContext = ActionMessage.PrepareContext;

ActionMessage.PrepareContext = context => {

oldPrepareContext(context);

PrepareContext(context);

};

ActionMessage.InvokeAction = InvokeAction;

}

public static void PrepareContext(ActionExecutionContext context) {

var contextAwareFilters = FilterManager.GetFiltersFor(context)

.OfType<IContextAware>()

.ToArray();

contextAwareFilters.Apply(x => x.MakeAwareOf(context));

context.Message.Detaching += (o, e) => { contextAwareFilters.Apply(x => x.Dispose()); };

}

}

## Implementing Filters

To simplify filters construction, use the base class for IExecutionWrapper which includes all the boilerplate code and provides the standard customization points:

public abstract class ExecutionWrapperBase : Attribute, IExecutionWrapper, IResult {

IResult inner;

event EventHandler<ResultCompletionEventArgs> CompletedEvent = delegate { };

public int Priority { get; set; }

IResult IExecutionWrapper.Wrap(IResult inner) {

this.inner = inner;

return this;

}

void IResult.Execute(ActionExecutionContext context) {

if(!CanExecute(context)) {

CompletedEvent(this, new ResultCompletionEventArgs { WasCancelled = true });

return;

}

try {

EventHandler<ResultCompletionEventArgs> onCompletion = null;

onCompletion = (o, e) => {

inner.Completed -= onCompletion;

AfterExecute(context);

FinalizeExecution(context, e.WasCancelled, e.Error);

};

inner.Completed += onCompletion;

BeforeExecute(context);

Execute(inner, context);

}

catch(Exception ex) {

FinalizeExecution(context, false, ex);

}

}

event EventHandler<ResultCompletionEventArgs> IResult.Completed {

add { CompletedEvent += value; }

remove { CompletedEvent -= value; }

}

protected virtual bool CanExecute(ActionExecutionContext context) {

return true;

}

protected virtual void BeforeExecute(ActionExecutionContext context) {}

protected virtual void AfterExecute(ActionExecutionContext context) {}

protected virtual void Execute(IResult inner, ActionExecutionContext context) {

inner.Execute(context);

}

protected virtual bool HandleException(ActionExecutionContext context, Exception ex) {

return false;

}

void FinalizeExecution(ActionExecutionContext context, bool wasCancelled, Exception ex) {

if(ex != null && HandleException(context, ex))

ex = null;

CompletedEvent(this, new ResultCompletionEventArgs { WasCancelled = wasCancelled, Error = ex });

}

}

## Sample Filters and Usage

/// <summary>

/// Provides asynchronous execution of the action in a background thread

/// </summary>

public class AsyncAttribute : ExecutionWrapperBase

{

protected override void Execute(IResult inner, ActionExecutionContext context)

{

ThreadPool.QueueUserWorkItem(state =>

{

inner.Execute(context);

});

}

}

//usage:

//[Async]

//public void MyAction() { ... }

/// <summary>

/// Allows to specify a "rescue" method to handle exception occurred during execution

/// </summary>

public class RescueAttribute : ExecutionWrapperBase

{

public RescueAttribute() : this("Rescue") { }

public RescueAttribute(string methodName)

{

MethodName = methodName;

}

public string MethodName { get; private set; }

protected override bool HandleException(ActionExecutionContext context, Exception ex)

{

var method = context.Target.GetType().GetMethod(MethodName, new[] { typeof(Exception) });

if (method == null) return false;

try

{

var result = method.Invoke(context.Target, new object[] { ex });

if (result is bool)

return (bool)result;

else

return true;

}

catch

{

return false;

}

}

}

//usage:

//[Rescue]

//public void ThrowingAction()

//{

// throw new NotImplementedException();

//}

//public bool Rescue(Exception ex)

//{

// MessageBox.Show(ex.ToString());

// return true;

//}

/// <summary>

/// Sets "IsBusy" property to true (on models implementing ICanBeBusy) during the execution

/// </summary>

public class SetBusyAttribute : ExecutionWrapperBase

{

protected override void BeforeExecute(ActionExecutionContext context)

{

SetBusy(context.Target as ICanBeBusy, true);

}

protected override void AfterExecute(ActionExecutionContext context)

{

SetBusy(context.Target as ICanBeBusy, false);

}

protected override bool HandleException(ActionExecutionContext context, Exception ex)

{

SetBusy(context.Target as ICanBeBusy, false);

return false;

}

private void SetBusy(ICanBeBusy model, bool isBusy)

{

if (model != null)

model.IsBusy = isBusy;

}

}

//usage:

//[SetBusy]

//[Async] //prevents UI freezing, thus allowing busy state representation

//public void VeryLongAction() { ... }

/// <summary>

/// Updates the availability of the action (thus updating the UI)

/// </summary>

public class DependenciesAttribute : Attribute, IContextAware

{

ActionExecutionContext \_context;

INotifyPropertyChanged \_inpc;

public DependenciesAttribute(params string[] propertyNames)

{

PropertyNames = propertyNames ?? new string[] { };

}

public string[] PropertyNames { get; private set; }

public int Priority { get; set; }

public void MakeAwareOf(ActionExecutionContext context)

{

\_context = context;

\_inpc = context.Target as INotifyPropertyChanged;

if (\_inpc != null)

\_inpc.PropertyChanged += inpc\_PropertyChanged;

}

public void Dispose()

{

if (\_inpc != null)

\_inpc.PropertyChanged -= inpc\_PropertyChanged;

\_inpc = null;

}

void inpc\_PropertyChanged(object sender, PropertyChangedEventArgs e)

{

if (PropertyNames.Contains(e.PropertyName))

{

Execute.OnUIThread(() =>

{

\_context.Message.UpdateAvailability();

});

}

}

}

//usage:

//[Dependencies("MyProperty", "MyOtherProperty")]

//public void DoAction() { ... }

//public bool CanDoAction() { return MyProperty > 0 && MyOtherProperty < 1; }

/// <summary>

/// Allows to specify a guard method or property with an arbitrary name

/// </summary>

public class PreviewAttribute : Attribute, IContextAware

{

public PreviewAttribute(string methodName)

{

MethodName = methodName;

}

public string MethodName { get; private set; }

public int Priority { get; set; }

public void MakeAwareOf(ActionExecutionContext context)

{

var targetType = context.Target.GetType();

var guard = targetType.GetMethod(MethodName);

if (guard== null)

guard = targetType.GetMethod("get\_" + MethodName);

if (guard == null) return;

var oldCanExecute = context.CanExecute;

context.CanExecute = () =>

{

if (!oldCanExecute()) return false;

return (bool)guard.Invoke(

context.Target,

MessageBinder.DetermineParameters(context, guard.GetParameters())

);

};

}

public void Dispose() { }

}

//usage:

//[Preview("IsMyActionAvailable")]

//public void MyAction(int value) { ... }

//public bool IsMyActionAvailable(int value) { ... }