What’s on Tap?

TBD: Introduction to this tutorial series.

# Happy Hour Agenda

Blurbs and links to the individual lessons

# Belly up to the bar

**Download the** **Cocktail** zip file [reference needed]

**Unblock the zip file**

You unblock the zip file or your application references to Framework assemblies will fail. Right-click the zip file, select “Properties”, and click the “Unblock” button at the bottom of the dialog:

****

Navigate to the HappyHour directory within the Samples folder. Inside you will find numerous numbered HappyHour folders. “HappyHour” is a multi-lesson tutorial during which you build a sample application called “Happy Hour”. Each numbered folder holds the state of the “*Happy Hour*” application as it was at the end of the corresponding lesson.

**Mixers** contains material (e.g., code, xaml, images) that you’ll incorporate in the application solution as you follow along with the tutorial.

You will reference additional libraries throughout the tutorial. You will add some references by pointing to assemblies included in the download; some (e.g., .NET and DevForce assemblies) you will reference in the Global Assembly Cache (**GAC**); some you will acquire via the **NuGet** package installer.

You should install the [NuGet](http://nuget.org/) extension to your Visual Studio if you haven’t done so already. It is an easy and increasingly popular way to download, reference, and update both .NET and 3rd party components from the web.

Lesson 1: Bare Bones MVVM

# Create a single-View app

We begin with a simple application consisting of a single **View** that has no Cocktail support at all. Our application has a *TextBox* and a button. The user types the name of a cocktail in the *TextBox*, clicks the button, and a message window pops up that says “**One [whatever-the-user-entered], coming right up!**”



With our eye on the future, we soon will over-engineer this example by writing it in Model-View-ViewModel (MVVM) style. But let’s start with just a View class.

The **01-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

# Beginnings

**File | New | Project | Silverlight Application | “HappyHour”**

* **Uncheck** “Host in a new Web site” as we won’t need a server for quite a while
* The **Silverlight Version** should be consistent. It can be Silverlight 4 or Silverlight 5.

**Add a *Views* folder** to the HappyHour project.

**Drag the *MainPage.xaml* file into the *Views* folder**

In Cocktail, the *MainPage* is known as a “View”. A real application will have a great number of views; we’re structuring this tutorial app to conform to [our vision of that future](#_Why_bother_with).

**Open MainPage.xaml** in Design view with split panel (Design/XAML) showing both the design canvas and the XAML.

**Rename the class, “*HappyHour.Views.MainPage*”**

<UserControl x:Class="HappyHour.Views.MainPage"

**Insert the following within the *<Grid>…</Grid>* tags**.

<StackPanel Margin=“8,0,0,8”>

    <TextBlock Text="It's HAPPY HOUR!" FontSize="36" TextAlignment="Center"/>

    <TextBlock Text="What are you having?" FontSize="18"/>

    <TextBox x:Name="DrinkName" Margin="0,8,0,8" />

    <Button  x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

</StackPanel>

**Double-click the Button** in the design canvas to generate a click handler. Visual Studio responds first by adding a “Click” attribute to the Button XAML:

<Button ... Click="AskForIt\_Click" ... />

Then Visual Studio shows the click handler in the code-behind which we **modify to be**:

private void AskForIt\_Click(object sender, RoutedEventArgs e)

{

    MessageBox.Show(

        string.Format(CultureInfo.CurrentCulture,

        "One {0}, coming right up!", DrinkName)); // don't do this in real app

}

You’ll need to add “*using System.Globalization;*” to support the *CultureInfo* parameter. It’s worth it to prepare your string formats for globalization.

**Open MainPage.xaml.cs** code-behind file.

**Update the namespace to “*HappyHour.Views*”**

### Build and run [F5]

Enter a drink name in the text box – be careful … it is way too early for serious drinking – and press the button. The app responds:

# C:\Users\Ward\Documents\Visual Studio 2010\Projects\Samples\Cocktail_101\Tutorial Images\01_HappyHour_run_1.png

# Make it prettier

We’re not graphic designers but even we can add some style to liven this up.

**Add | New Folder | “assets”** …

This folder will hold a variety of “assets” to beautify your application. In a big application we’d have subfolders dedicated to different types of assets but for now we’ll keep it flat.

We’ll add three assets located within the Cocktail Mixers folder. Because they’re in separate subdirectory, you’ll do the following procedure three times:

* **Add | Existing Item**
* **Browse** to the pertinent ***HappyHour / Mixers /*** subdirectory
* **Select** the asset
* **Click** the “Add” button (or “Add As Link” if you prefer).

The three assets are:

* **HappyHourStyles** - a *ResourceDictionary* of styles in the **styles** subdirectory.
* **SegoeWP.ttf** - a “True Type” Metro font in the **fonts** subdirectory.
* **TypicalCoctail.png** - a colorful image in the **images** subdirectory.

**Open App.xaml**

Add the style *HappyHourStyles* *ResourceDictionary* to our application-wide resources

<Application.Resources>

    <!-- Resources scoped at the Application level should be defined here. -->

    <ResourceDictionary>

        <ResourceDictionary.MergedDictionaries>

            <ResourceDictionary Source="assets/HappyHourStyles.xaml"/>

        </ResourceDictionary.MergedDictionaries>

    </ResourceDictionary>

</Application.Resources>

**Return** to **MainPage.xaml**

**Set d:DesignWidth="550"** at the top of the XAML so we have some room.

We’re ready to spruce up the View. We make a two-column grid with the form on the left and an arresting image on the right. We add styles to the form elements.

The result is as follows (you can just paste it in … I know you will).

<Grid x:Name="LayoutRoot" Background="White">

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width="Auto"/>

        <ColumnDefinition Width="Auto"/>

    </Grid.ColumnDefinitions>

    <StackPanel Margin=“8,0,0,8”>

        <TextBlock Text="It's HAPPY HOUR!" Style="{StaticResource TitleTextBlock}" />

        <TextBlock Text="What are you having?" Style="{StaticResource QuestionTextBlock}" />

        <TextBox x:Name="DrinkName" Margin="0,8,0,8" />

        <Button  x:Name="AskForIt" Content="Ask for it" Click="AskForIt\_Click" Margin="0,0,0,4"/>

    </StackPanel>

    <Image Source="/HappyHour;component/assets/cocktail\_logo\_big.png" Grid.Column="1 />

</Grid>

**Build and run again [F5]**



Still too early to celebrate but we’re feeling the first stirrings of excitement.

**Close all windows** (Alt-W, L)

# Add a MainPageViewModel

Most Silverlight and WPF application Views are supported by a ***ViewModel***. We’ll write a bare-bone *ViewModel* without the aid of a supporting framework.

**Add a *ViewModels* folder** to the *HappyHour* project

**Add | New Item | Class | “MainPageViewModel”**

**Add a *DrinkName* property**:

public string DrinkName { get; set; }

**Add an *AskForIt* method** as follows:

public void AskForIt()

{

    MessageBox.Show(

        string.Format(CultureInfo.CurrentCulture,

        "One {0}, coming right up!", DrinkName)); // don't do this in real app

}

You’ll have to add "*using System.Globalization;*" to access CultureInfo.

**Open *MainPage.xaml*** and add a binding to the *TextBox*

<TextBox x:Name=“DrinkName” Margin="0,8,0,8" Text="{Binding DrinkName, Mode=TwoWay}" />

**Open the *MainPage.xaml.cs*** code-behind

**Add** *using HappyHour.ViewModels;*

**Revise** the class implementation as follows:

public partial class MainPage : UserControl

{

    private readonly MainPageViewModel \_viewModel;

    public MainPage()

    {

        InitializeComponent();

        DataContext =\_viewModel = new MainPageViewModel();

    }

    private void AskForIt\_Click(object sender, RoutedEventArgs e)

    {

        \_viewModel.AskForIt();

    }

}

Observe that:

* The *View* instantiates the *ViewModel* in the constructor
* The constructor sets the *View*’s *DataContext* with the *ViewModel* so that the “DrinkName” *TextBox* binds to the *ViewModel*’s *DrinkName* property at runtime.
* The *View* records the ViewModel instance in a typed variable (*\_viewModel*) for later reference.
* The “*AskForIt\_Click*” method now re-routes its previous behavior to the *ViewModel*.

### Build and run [F5]

Confirm that the application runs exactly as it did before.

# Enable the Button Conditionally

Right now the button is always enabled even when the drink name *TextBox* is empty. Click the button and it just says “One, coming right up!”



**What** is coming right up? We want the message to appear only when the app knows what to deliver. We will disable the button when the *TextBox* is empty and enable it when the *TextBox* has a value.

**Return to the MainPageViewModel** and add the *CanAskForIt* guard property:

public bool CanAskForIt

{

    get { return !String.IsNullOrWhiteSpace(DrinkName); }

}

**Return to the MainPage.xaml** and modify the button so that its *IsEnabled* property is bound to the ViewModel’s *CanAskForIt* property.

<Button x:Name="AskForIt" Content="Ask for it" Click="AskForIt\_Click"

IsEnabled="{Binding CanAskForIt}" Margin="0,0,0,4"/>

**Return to the MainPageViewModel**

Now that the Button’s *IsEnabled* property is bound to the *CanAskForIt* property, the button will be disabled initially because there is no *DrinkName* text.

The *Button* won’t become enabled until (a) the *ViewModel* tells the *View* that the *CanAskForIt* property has changed and (b) the property’s value becomes *true*. If you ran the application right now, the button would be disabled and would stay disabled no matter what we typed in the drink name *TextBox*. If you entered text and tested the *CanAskForIt* property in the debugger, it would return *true*. But the View doesn’t know that … and won’t know that … until the ViewModel notifies it.

## Enable ViewModel change notification

Make *MainPageViewModel* inherit from *System.ComponentModel.INotifyPropertyChanged* and implement that interface as follows:

public class MainPageViewModel : INotifyPropertyChanged

{

// ... elided ...

    public event PropertyChangedEventHandler PropertyChanged;

    protected void NotifyOfPropertyChange(string propertyName)

    {

        var handlers = PropertyChanged;

        if (null == handlers) return; // no one is listening

        handlers(this, new PropertyChangedEventArgs(propertyName));

    }

}

Now the View will refresh a property binding when we call *NotifyOfPropertyChange* with the name of the property to refresh. In this case, we want the View to update when the state of *CanAskForIt* changes.

*CanAskForIt* changes when the *DrinkName* changes. We need the setter of the *DrinkName* property to raise the *PropertyChanged* event for *CanAskForIt*. **Rewrite the DrinkName property** as follows:

private string \_drinkName;

public string DrinkName

{

    get { return \_drinkName; }

    set {

        \_drinkName = value;

        NotifyOfPropertyChange("CanAskForIt");

    }

}

### Build and run [F5]

The button starts disabled. Type the name of a drink in the *TextBox* and **Tab out** of the *TextBox*. The button is enabled. When you click it, the *MessageBox* appears.

## An unpleasant experience

We’d really like to enable or disable the button as the user types in the *TextBox*. At the moment there are no visual cues that the button could be pressed and with no other enabled controls on the screen that are clickable, the user could think the application is frozen.

The problem is that the *TextBox* binding doesn’t update the ViewModel’s *DrinkName* property until the *TextBox* ***loses focus***. In WPF you can set the *UpdateSourceTrigger* of the *TextBox* binding to “*PropertyChanged*” so that the *DrinkName* property updates ***as the user types***.

Let’s add *UpdateSourceTrigger* option to the TextBox binding and set the option to update the ViewModel’s *DrinkName* property as the user types.

<TextBox x:Name="DrinkName" Margin="0,8,0,8"

Text="{Binding DrinkName, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />

The *UpdateSourceTrigger* option doesn’t exist in Silverlight 4. You’d have to either wait for the user to tab out of the *TextBox* or do some fancy keystroke tracking in the code-behind to detect when the user is typing. Fortunately, it’s been added to Silverlight 5.

# Last Call

We’re done for now. It’s time for a refreshing drink and a review of what we’ve done.

We’ve created a simple, one View Silverlight application that follows the Model-View-ViewModel (MVVM) pattern without the aid of any framework.

Perhaps we should have stuck with our initial version that kept all logic in the View code-behind. It took work to get to MVVM and we have little to show for it so far. The author is willing to bet the benefits of MVVM will become apparent … but feels the anticipatory investment of effort should be lower. In the next lesson, we’ll see what Cocktail can do to make writing in MVVM style much easier.

# Ask the Mixologist

Our first lesson is finished. Feel free to move on directly to the next one. This “Ask the Mixologist” section is an optional digression from the lesson’s main course to related points of interest.

## What is a View?

A **View** presents information and accepts input from a human user. A View is typically manifested as a rectangular area of on-screen real estate containing visual controls (TextBlocks, *TextBox*es, Buttons, etc.).

The boundaries of the view are up to you. It can be as small as a single control or fill the entire display. Views can contain other views (“view composition”). Views can appear in the main window (“the shell”) or in a popup window.

## What is a ViewModel?

Most Views display and update data. There can be a lot of code involved in retrieving the data, forwarding data values to the user controls, validating user input, responding to user gestures (e.g., button clicks and mouse-overs), and saving changed data.

You can put all of this logic into the View class’s code-behind. But most developers believe their code is easier to maintain when they move most of these mechanics to a separate helper class called the **ViewModel**.

Such separation allows the developer of the View to concentrate on the visual design and the immediate interactions between the View and the user without simultaneously addressing the non-visual dependencies such as where data come from or the business and application logic. Those concerns are handled separately by the developer of the ViewModel who can proceed without worrying about the artful design of an expressive and effective View. This separation is helpful even when the developer of the View and the developer of the ViewModel are the same person.

The division of responsibilities between the View and the ViewModel is central to the “Model-View-ViewModel” (**MVVM**) pattern [reference needed] favored by Silverlight and WPF developers.

## What is the difference between MVVM and MVC?

If you are familiar with **MVC** (“Model-View-Controller”) than you already have a grasp of the essential motivation behind the **MVVM** (“Model-View-ViewModel”) pattern: the separation of the purely visual representation of the View from the non-visual state and behaviors that make the View “work”.

The *ViewModel* in MVVM plays the same role as the *Controller* in MVC. It acts as the non-visual helper class that makes the View “work”. The differences between *ViewModel* and *Controller* revolve around how the helper class supports the View.

A *Controller* in MVC has a more direct awareness and involvement in the preparation and servicing of the View even when that awareness is somewhat abstracted through an interface.

Silverlight and WPF rely on a declarative **data binding** system that maps controls in the View to corresponding members of the *ViewModel* without either the View or the ViewModel having knowledge of the other.

The core principle - separating the presentation into a View and non-View helper – is what really matters. The choice between MVC and MVVM is usually based on the facilities available in the client technology you chose. Silverlight and WPF’s rich, bidirectional data binding incline toward the MVVM pattern.

## Where is the Model in this MVVM app?

The Model of a typical business app is a coherent set of classes that hold application data and implement the business rules that govern those data. “DrinkOrder”, “Product”, and “Order” are names of classes you might expect in an order management system.

There is no such Model in this example. Stay with us; we’ll get to the Model a few lessons from now.

## Why bother with the folders?

I had you create “Views” and “ViewModels” folders right from the start. At the moment you only have one class in each folder. Isn’t this over-engineering?

It certainly would be if we expected only one *View* and one *ViewModel* in our application. Of course MVVM and Cocktail itself would be over-engineering in a single screen application.

If you postpone this step, you won’t realize you have a problem until the project is cluttered with twenty or thirty *View/ViewModel* pairs. By then the logistics of moving files and repairing misaligned namespaces will be truly painful.

You aren’t speculating that you might have a lot of views someday. It’s a near-term certainty. In this case, an ounce of prevention really is worth a pound of cure. Do it now; thank me later.

## Why is it a bad idea to call *MessageBox* in the ViewModel?

A comment in the *AskForIt* method warns us that we should not call *MessageBox.Show* in the ViewModel of a real application.

The problem isn’t that we are using a *MessageBox* in our UI – that could be a reasonable design choice. The problem is that we are referring to the *MessageBox* class in our ViewModel. That’s a bad idea for two reasons:

*Principle*: a ViewModel should avoid dependence on specific UI controls. The View –not the ViewModel - determines what the user sees and how the user interacts with the application. In this case, the ViewModel dictates the user experience, preventing the UI designer from doing his/her job which is to find the best and most consistent way for users to see and respond to information.

*Practical*: By baking the *MessageBox* class into the ViewModel we have made the ViewModel difficult to test. To test if calling the *AskForIt* method does the right thing, the test harness would have to spin up a *MessageBox* and our test code would have to find and inspect that *MessageBox* to determine that it presented the expected message. That’s too much work just to find out if our *AskForIt* method is saying “One Daiquiri, coming right up!” when the drink name is “Daiquiri”. When testing is hard, we don’t do it … with predictable adverse consequences.

A better approach is to define and call upon an interface of a message displaying component. An appropriate implementation of that interface would be provided to the ViewModel at test and runtime. We may take up an illustration of this alternative in a future lesson.

## Could we have bound the button to an ICommand?

Silverlight and WPF abstract and unify command combinations such as *AskForIt* and *CanAskForIt* with a version of “*Commanding*” based on the *System.Windows.Input.ICommand* interface.

If we chose this path we would

* Add a generalized *Command* class to our solution that implements *ICommand*
* Modify the *MainPageViewModel* to
  + expose an *AskForItCommand*
  + extend the *DrinkName* property setter to *RaiseCanExecuteChanged* when the propertychanges
* In *MainPage*, bind the button’s *Command* property to the ViewModel’s *AskForItCommand*
* Delete the obsolete *AskForIt\_Click* handler from the *MainPage* code-behind.

We’ve included an example of this approach in the ***01-HappyHour*** sample that accompanies this lesson.

The benefit is that we’ve reduced the size of the code-behind by the click handler. Minimizing code in code behind is a priority for MVVM developers.

But is it worth the effort? We must write the ViewModel *AskForIt* and *CanAskForIt* members regardless. The button click handler, which is trivially easy to read and write, merely delegates to the ViewModel. The *Command* object is extra complexity and you can only use this kind of Command binding approach for buttons; we want commanding for other kinds of events too (e.g., selection in a *ComboBox*). Why bother?

Lesson 2: An MVVM Cocktail

In Lesson 1 we created a simple, one-View Silverlight application. You enter the name of a drink, press the button, and the application promises to deliver it to you.



We implemented it first with a single *MainPage* class. Then, in keeping with the **MVVM** (“Model-View-ViewModel”) pattern, we factored out a *MainPageViewModel* class from the *MainPage*’s code-behind.

In this lesson, we’ll identify the many hazards and complexities involved in our original design. Then we will see what Cocktail can do to alleviate them.

The **02-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

# Pick up where we left off

You could continue with the solution from Lesson 1 exactly as you left it. Alternatively, you can start over with the original contents of the folder “**01-HappyHour**” from the tutorial zip file.

**Build and run [F5]** to confirm it still works. Type into the *TextBox*, tab out, and click the button to see the behavior depicted above.

# A “Lesson 1” Code Review

The “Lesson 1” solution is still in its pre-Cocktail state. Right now there is a lot of MVVM-inspired code that introduces plenty of complexity for no obvious benefit.

You have a choice. You can stick with me in this section as I tear this code apart. Or you can skip to the next section and get on with the business of fixing it.[Link to “Refactor”]

Start with the ***MainPageViewModel*** itself … which is inevitable when you follow MVVM. It’s more than 50 lines. Did we cut 50 lines somewhere else? No we did not. The *MainPage.xaml.cs* code-behind is about the same length (~20 lines) it was *before* we added the *ViewModel*.

We could cut the 5 lines devoted to the disused *AskForIt\_Click* handler

The *DrinkName* property is a stand-in for the kind of Model object that you would display in a typical business application. It could have been a one-line auto-property were it not for the need to notify the *View* when the drink name changes. Now it takes nine lines just so it can notify the UI via *NotifyOfPropertyChange* when the property is set. We’ll be stuck with this same 9-fold expansion for each of the many properties we add to this ViewModel as it grows to support a real use case.

We also carry the eight lines at the bottom devoted to *INotifyPropertyChanged*. You don’t want to repeat that in every ViewModel so we should expect to push that into a base class … one more thing to remember.

The *AskForIt* and *CanAskForIt* members carry their own weight; you’d implement them in roughly the same way whether in the *ViewModel* or in the *View* code-behind. We can live with that.

But what if you choose to use native Silverlight commanding as we did in the **01-HappyHour sample?** That will cost you another nine lines for the *AskForItCommand* property. Worse, this expedient exists ***solely*** to satisfy the XAML binding technology. That’s annoying.

Another source of complexity lurks in those data bindings. Look again at the *TextBox* binding in the *MainPage*.

<*TextBox* BorderThickness="1" BorderBrush="Blue" Margin="4"

         Text="{Binding DrinkName, Mode=TwoWay}"/>

Would you have remembered to write “*Mode=TwoWay*”? I often forget. What if you forgot? Would you notice it was missing?

Remove it now and run the application again. Type anything into the *TextBox* and tab out. The button fails to enable this time. Do you know why?

Confirm that the ViewModel’s *DrinkName* property remains empty no matter what you enter in the *TextBox*. The value in the *TextBox* isn’t passed along to the ViewModel. By rule, the *CanAskForIt* guard property can never return *true* when the *DrinkName* is empty so the button never enables.

Nothing is wrong from the Silverlight perspective. You won’t get an exception; there will be no binding failure reported in the output window. The default *Mode* in Silverlight is “*OneWay*”, which translates to “you don’t care about user input”. Although that’s not what you expected – not what anyone expects from *TextBox* data entry – it’s OK with Silverlight.

How many development hours will you waste trying to figure out why the button isn’t enabled … all because of this unfortunate default?

By my calculation, we’ve doubled or tripled the size of the *View* logic and certainly doubled the number of classes. If this trend continues for each of the many *View*s we expect to add to our application, we’re looking at huge bloat potential. It wouldn’t actually be *that* bad. But it’s bad enough.

Our handwritten MVVM implementation demands too much attention to picayune detail. There is too much “ceremony code” – too much fat – and it all gets between us and the realization of our application’s purpose.

Cocktail can trim away a lot of the fat. Let’s refactor this application to use Cocktail.

## Add Cocktail Project Reference

Most application developers will reference the Cocktail assembly they keep in a dependency directory such as a “Lib” folder. In this tutorial sample, we’ll include the Cocktail Silverlight project itself so that (a) it’s handy when we want to inspect the referenced source code and (b) we work with a freshly built version of Cocktail.

**Add | New Solution Folder | “Supporting Projects”**

A Cocktail project isn’t really part of the HappyHour application. We’re only including it for reference. To make that point crystal clear, we’ll tuck it under this Visual Studio Solution folder.

And under this folder, …

**Add | Existing Project | navigate up to the Cocktail folder | Cocktail.SL**

**Build the project**.

If it doesn’t build successfully, check the references. Can the Cocktail project find its three referenced “IdeaBlade” assemblies? If not, you may need to [install DevForce](http://drc.ideablade.com/xwiki/bin/view/Documentation/cocktail-getting-started#HInstallDevForce) (included with your Cocktail download) which is a prerequisite for Cocktail applications. You must have installed DevForce ***after installing Silverlight 5*** or Cocktail won't find the DevForce Silverlight libraries. If you installed DevForce first, the easiest resolution is to reinstall DevForce.

## Add project dependencies to *Caliburn.Micro* and *MEF*

**Add project reference** **to the *Caliburn.Micro*** Silverlight assembly, located in the Cocktail download; find it by navigating from the top to ***src/packages/Caliburn.Micro.1.3.1/lib/***

From there pick either ***SL4/Caliburn.Micro.dll*** or ***SL5/Caliburn.Micro.dll*** depending upon which version of Silverlight you’re using.

**Add project reference** **to *System.ComponentModel.Composition*** also known as “MEF”.

A Cocktail application discovers and manages components through a mechanism widely known as “Dependency Injection” or “IoC” (“Inversion-of-Control). In Cocktail we rely upon “MEF” (the “Managed Extensibility Framework”) for dependency injection which is why we added this library.

## [*Export*] the *MainPageViewModel*

We’re using MEF version 1 which means we have to mark certain of our classes with attributes so that MEF can find them. We’ll go into more detail in a later lesson. For now, know that you should add the [*Export*] attribute to *ViewModel*s such as *MainPageViewModel*:

[Export]

public class MainPageViewModel : INotifyPropertyChanged {...}

## Add a Bootstrapper

A bootstrapper is the start-up code that prepares an application’s execution environment and initiates the first step. Every application you’ve ever written has a bootstrapper of some sort. The “*Silverlight Application*” template locates nearly 60 lines of bootstrapping code in the *App.xaml.cs*.

Cocktail has its own startup code in a *FrameworkBootstrapper* class that you should inherit from and extend for your needs. It incorporates the standard *App.xaml.cs* boilerplate, configures the application and loads the view associated with your topmost, root *MainPageViewModel*.

**Add | New Item | Class | “AppBootstrapper”**

**Inherit** from *Cocktail.FrameworkBootstrapper<MainPageViewModel>*

public class AppBootstrapper : Cocktail.FrameworkBootstrapper<MainPageViewModel> { }

The *MainPageViewModel* type specification tells Cocktail to construct an instance of *MainPageViewModel*, construct an instance of its companion *MainPage* view, bind them together, and then display the *MainPage* as the *RootVisual* – the main window – of the Silverlight application.

**Open App.Xaml** and **insert a bootstrapper resource**; the resulting file should look like this:

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

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

             xmlns:local="clr-namespace:HappyHour"

             x:Class="HappyHour.App"

             >

    <Application.Resources>

        <!-- Resources scoped at the Application level should be defined here. -->

        <ResourceDictionary>

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

            <ResourceDictionary.MergedDictionaries>

                <ResourceDictionary Source="assets/HappyHourStyles.xaml"/>

            </ResourceDictionary.MergedDictionaries>

        </ResourceDictionary>

    </Application.Resources>

</Application>

By adding our *AppBootstrapper* as an *App.xaml* resource we ensure that our application is properly configured before the Shell is displayed whether in a running application or in a design tool such as **Blend** or Visual Studio’s “***Cider***”.

We no longer need – or want – the code in the *App.xaml* code-behind.

**Open App.xaml.cs** and **remove everything** inside the *App* class definition. All we need is a constructor that calls *InitializeComponent()*.

using System.Windows;

namespace HappyHour

{

    public partial class App : Application

    {

        public App()

        {

            InitializeComponent();

        }

    }

}

So far we’ve added one new class (*AppBootstrapper*) and compensated by removing fifty lines of *App.xaml.cs* code-behind. That’s a fair trade.

**Build and run [F5]**

The application should run almost exactly as before. We haven’t improved anything; but we haven’t broken much of anything either.

You may notice that the *MessageBox* appears twice. The *AppBootstrapper* has engaged behavior that duplicates the button click handler; we’ll deal with that shortly.

## Trim the View fat

Over the next series of steps we will whittle away the MVVM plumbing code that we no longer.

We’re about to remove the explicit bindings between the View and the ViewModel. We’re removing them because we want convention-based binding to take over. This is our choice. But remember that explicit binding remains an option. Explicit binding always trumps conventional binding. Other factors (e.g., using Blend to design the view with data) may argue for explicit data binding even when conventional binding would do the job at runtime.

**Open MainPage.xaml**

**Remove the binding** from the ***TextBox*** … as shown here:

<TextBox x:Name="DrinkName" Style="{StaticResource TitleTextBlock}"/>

**Remove the bindings** from the **Button** … as shown here:

<Button x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

**Delete x:Name="Layout"** attribute from the ***Grid***; this grid name is harmless but we favor cleanliness.

After these changes, the *Grid* looks like this:

<Grid>

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width="Auto"/>

        <ColumnDefinition Width="Auto"/>

    </Grid.ColumnDefinitions>

    <StackPanel Margin=“8,0,0,8”>

        <TextBlock Text="It's HAPPY HOUR!" Style="{StaticResource TitleTextBlock}" />

        <TextBlock Text="What are you having?" Style="{StaticResource QuestionTextBlock}" />

        <TextBox x:Name="DrinkName" Margin="0,8,0,8" />

        <Button  x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

    </StackPanel>

    <Image Source="/HappyHour;component/assets/cocktail\_logo\_big.png" Grid.Column="1 />

</Grid>

### Build and run [F5]

We’ve removed all bindings and the “Click” event wiring. We have named the elements that we want to bind to the ViewModel, using the standard “x:Name” attribute supported by every XAML element.

The elementnames we choose follow naming conventions that enable Cocktail to **map the controls to corresponding properties and methods of the ViewModel** and build the appropriate bindings dynamically at runtime.

We no longer have to remember pesky details such as “*Mode=TwoWay*”; Cocktail assumes that is what we intend when we bind to a *TextBox*. There are other binding attributes we haven’t mentioned – the attributes having to do with validation for example. Cocktail adds them too.

At the start, the “DrinkName” *TextBox* is empty and the button is disabled. Type a single character in the *TextBox* and the button lights up. You didn’t have to tab out of the control. Cocktail listens for changes to the *TextBox* contents and updates the *DrinkName* property immediately … which signals the View to re-read the *CanAskForIt* property … and enable/disable the button accordingly.

We get the expected behavior with less XAML and no new code.

## Trim the ViewModel fat

**Open the *MainPageViewModel***

**Add *using Caliburn.Micro;***

**Inherit from *Screen*** instead of *INotifyPropertyChanged*

[Export]

public class MainPageViewModel : Screen

Most of the ViewModels you write for a Cocktail application will inherit from *Caliburn.Micro.Screen*. *Screen* has many capabilities and we’ll draw upon several of them in the course of this tutorial. Right now we benefit from its implementation of *INotifyPropertyChanged*.

**Delete the implementation of *INotifyPropertyChanged***

**Delete everything having to do with Command**

**Delete the *MainPage* code-behind** (“MainPage.xaml.cs”).

We don’t need it. We don’t need the *AskForIt\_Click* handler because Cocktail wires the button directly to the ViewModel’s *AskForIt* method. We don’t need to set the View’s *DataContext* because Cocktail does that. We don’t need to invoke the View’s *InitializeComponent* method; Cocktail does that too.

**Delete the Command.cs file** in Solution Explorer

**Build and run [F5]** to confirm that it still runs.

# A clean View/ViewModel pair

We are finished refactoring our *View* and *ViewModel* and are ready to assess the consequences

The substance of *MainPageViewModel.cs* has shrunk to roughly 20 lines, down from 40+:

[Export]

public class MainPageViewModel : Screen

{

    private string \_drinkName;

    public string DrinkName

    {

        get { return \_drinkName; }

        set

        {

            \_drinkName = value;

            NotifyOfPropertyChange("CanAskForIt");

        }

    }

    public bool CanAskForIt

    {

        get { return !String.IsNullOrWhiteSpace(DrinkName); }

    }

    public void AskForIt()

    {

    MessageBox.Show(

        string.Format(CultureInfo.CurrentCulture,

        "One {0}, coming right up!", DrinkName)); // don't do this in real app

    }

}

The *MainPage* no longer has a code-behind file. Its entire substance is captured in the top-level *Grid*:

<Grid>

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width="Auto"/>

        <ColumnDefinition Width="Auto"/>

    </Grid.ColumnDefinitions>

    <StackPanel Margin=“8,0,0,8”>

        <TextBlock Text="It's HAPPY HOUR!" Style="{StaticResource TitleTextBlock}" />

        <TextBlock Text="What are you having?" Style="{StaticResource QuestionTextBlock}" />

        <TextBox x:Name="DrinkName" Margin="0,8,0,8" />

        <Button  x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

    </StackPanel>

    <Image Source="/HappyHour;component/assets/cocktail\_logo\_big.png" Grid.Column="1" />

</Grid>

The separation of responsibilities between *View* and *ViewModel* is now crisp and clear. The data and logic are in the *ViewModel*; the UI widgets are in the *View*’s XAML. Neither class refers directly to the other; they are linked only by name.

The noise of *ViewModel* instantiation, commanding and data binding is low compared to our first homebrew MVVM attempt.

The implementation of the ViewModel’s *DrinkName* property remains disturbing, requiring nine lines when one or two should suffice. Were it not for this excess, the View/ViewModel combination would be smaller than the original View-only implementation.

There is no simpler way to write a notifying property native .NET … although we are working on an AOP solution that would reduce this example to two lines. Stay tuned.

# Last Call

We started with a handcrafted MVVM-style application from Lesson 1. We found it bloated with tedious code in C# and XAML that we think a framework ought to figure out on its own. By introducing Cocktail and following some simple, easy to remember conventions, we trimmed away the excess manual coding leaving classes whose size and complexity are commensurate with their contributions to the application.

# Ask the Mixologist

This lesson is finished. Feel free to move on directly to the next lesson. This “Ask the Mixologist” section is an optional digression from the lesson’s main course to related points of interest.

## Should I delete a View code-behind that does nothing?

When the code-behind consists entirely of a constructor that calls [*InitializeComponent*](#_Where_is_InitializeComponent?) it isn’t performing any useful function. Cocktail (Caliburn) calls *InitializeComponent* for us; there is no good reason to call it twice, even if doing so is harmless.

You don’t have to delete this file; I prefer to delete it for the same reason that I prefer to delete all “do nothing” files: dead code waste time and interfere with my ability to understand what the application is actually doing.

When I see *SomeView.xaml.cs* in Solution Explorer, I have to assume that it could contain important presentation logic. I’ll probably open that file twenty times over the life of the application, only to find that it does nothing at all. Multiply that futile exercise by a few hundred views to estimate frustration in time and money.

If I need it again, I’ll create it again … as we show in a later lesson, “Talking to the View”.

## Where is *InitializeComponent*?

The build process generates a partial class file for every XAML class. It’s hidden from view but you can find it in a project subdirectory, e.g., “*..\HappyHour\obj\Debug\ MainPage.g.cs*”.

Lesson 3: Conventions & Diagnostics

In Lesson 2 we revised our simple, one-View Silverlight application to with Cocktail’s interpretation of the **MVVM** (“Model-View-ViewModel”) pattern. We saw that Cocktail made many decisions on our behalf based largely on our observance of its default naming conventions.

In this lesson, we’ll look more closely at the conventions we’ve relied upon so far. Conventions are fine … as long as you follow them. If you inadvertently break with the conventions, your application can misbehave.

The **03-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

# Pick up where we left off

We won’t make any permanent changes to the application in this lesson. We’ll make temporary breaking changes so that we can see what trouble looks like and how to recover from common problems. At the end, we should be right back where we were at the end of “**02-HappyHour**”. For convenience we’ve provided a “**03-HappyHour**” for you to play with

**Build and run [F5]** to confirm it still works. After typing into the *TextBox* and clicking the button you should see.



# Coding by convention

The essence of the Happy Hour application is confined to two classes, the *MainPage* and its companion *MainPageViewModel*. The *MainPage* is a few lines of XAML with no code-behind:

<Grid>

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width="Auto"/>

        <ColumnDefinition Width="Auto"/>

    </Grid.ColumnDefinitions>

    <StackPanel Margin=“8,0,0,8”>

        <TextBlock Text="It's HAPPY HOUR!" Style="{StaticResource TitleTextBlock}" />

        <TextBlock Text="What are you having?" Style="{StaticResource QuestionTextBlock}" />

        <TextBox x:Name="DrinkName" Margin="0,8,0,8" />

        <Button  x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

    </StackPanel>

    <Image Source="/HappyHour;component/assets/cocktail\_logo\_big.png" Grid.Column="1" />

</Grid>

The *MainPageViewModel* is just a handful of public members with no apparent “view awareness”:

[Export]

public class MainPageViewModel : Screen

{

    public string DrinkName { get { ... }  set { ... }  }

    public bool CanAskForIt { get { ... } }

    public void AskForIt() { ... }

}

The economy of this approach stems from Cocktail’s ability to [connect the *MainPageViewModel* to the *MainPage*](#_How_did_it) view and bind the *View*’s controls to the *ViewModel*’s members … all by means of commonplace [naming conventions](#_What_are_the):

* *TextBox* “DrinkName” to *MainPageViewModel.****DrinkName***
* *Button* “AskForIt” to *MainPageViewModel.****AskForIt*** and *MainPageViewModel.****CanAskForIt***

We gave Cocktail an opportunity to do its thing by constructing a bootstrapper as a top level resource in the *App.xaml*:

<Application.Resources>

    <ResourceDictionary>

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

...

    </ResourceDictionary>

</Application.Resources>

The *AppBootstrapper* itself doesn’t look like much:

public class AppBootstrapper : Cocktail.FrameworkBootstrapper<MainPageViewModel> { }

That’s all it takes. The base *FrameworkBootstrapper* can:

* find the *MainPage* class that corresponds to the root *MainPageViewModel*
* compose instances of both classes and set the *DataContext* of the *MainPage* to the *MainPageViewModel*.
* inspect the instantiated *View*, gathering the “*x:Name*” values of UI controls.
* match these controls to compatibly named members of the *ViewModel*.
* bind the controls to these ViewModel members based on wiring rules specific to each type of control. For example, it binds the *TextBox*’s *Text* property to the ViewModel’s *DrinkName* property and binds the *Button*’s click behavior to the ViewModel’s *AskForIt* method and *CanAskForIt* guard property.

These are pretty much the same steps we performed manually in the code and XAML of our initial homebrew MVVM application back in Lesson 1. The beauty of the convention-based approach is that we don’t have to write that gunk any more. We can concentrate on the important stuff and let Cocktail do the grunt work.

Let’s examine the conventions in use in our example.

# The View/ViewModel matching convention

The first noteworthy convention matches the *ViewModel* to a *View*. In Cocktail applications we generally take what’s called a “*ViewModel First*” approach: we identify a type of *ViewModel* to compose and let the Caliburn Micro framework find, compose, and configure a corresponding *View*.

Per the default convention we take the name of the ViewModel, “*MainPageViewModel*”, and strip out the word “*Model*” to get “*MainPage*”. Then we look *in the same assembly* for a class with that name.

What if there is no matching view? Let’s create that problem and see what happens.

**Open *MainPage.xaml***

**Rename** its classto “**BadMainPage**”

<UserControl x:Class="HappyHour.BadMainPage"

**Rebuild and run [F5]**

The browser launches as before but instead of the application screen as we know it, we see this instead:



Cocktail is looking for a *View* to go with the *MainPageViewModel*. It can’t find one so Cocktail substitutes a default “Missing View” consisting of a single *TextBlock* with message text.

You may not always understand why Cocktail can’t find the matching *View* but at least you know what *ViewModel* prompted the search.

**Restore “BadMainPage”** **to “MainPage” and make sure it works**

<UserControl x:Class="HappyHour.MainPage"

Sometimes you have to clean the solution first to nudge Visual Studio into accepting these changes.

# Control binding conventions

Cocktail tries to associate the name of each control – the value of the control’s “x:Name” attribute – with a public member of the ViewModel. We have two named *MainPage* controls: the *TextBox* called “DrinkName” and the *Button* called “AskForIt”.

Cocktail associates those control names with the *DrinkName* property and the *AskForIt* method of the *MainPageViewModel* as you would expect. Cocktail also realizes that the *CanAskForIt* guard property determines whether or not the *AskForIt* method is allowed to be called at the moment; it then wires the *IsEnabled* state of the button accordingly.

As your application evolves there is a good chance you will add, remove, and change *ViewModel* member names. You won’t always remember to keep the *View* in sync. Coordination failures are even more likely when you add team members and divide *View* and *ViewModel* design responsibilities among those members.

Let’s see what happens when we make a few destructive changes.

**Remove the *x:Name*** attribute from the *TextBox* in ***MainPage***. We’re simulating a mistake in the XAML.

<TextBox BorderThickness="1" BorderBrush="Blue" Margin="4" />

**Rename the *AskForIt*** method to ***SockItToMe***in ***MainPageViewModel***. We’re simulating a name change in the *ViewModel* that wasn’t propagated to the *View* XAML.

public void SockItToMe() // AskForIt()

**Build and run [F5]**

The application compiles and runs. It appears as expected but misbehaves:

* The button is enabled whether or not there is text in the *TextBox*.
* Clicking the button does nothing.

The application didn’t throw an exception and there are no visual indications on screen or in any Visual Studio window to indicate that anything is amiss.

We know what is wrong of course. But you can imagine how frustrating this could be when you have more complicated views and many of them.

Don’t fix it yet. Leave it broken while we explore diagnostic remedies.

# Diagnose the configuration

**Run it again [F5]**

**Open the Visual Studio Output window** while the application is running.

Cocktail has installed a logger, the ***DefaultDebugLogger***, to track many aspects of your applications behavior. The log messages appear in the Output window and look something like this:

0 : 1/16/2012 5:12:45 PM : : IdeaBlade.Core.IdeaBladeConfig:Initialize : Initializing configuration ...

... More logs relating to IdeaBlade ... ignore for now ... focus on Cocktail.DefaultDebugLogger:Log

10 : ViewModelBinder INFO: Binding HappyHour.Views.MainPage and HappyHour.ViewModels.MainPageViewModel.

11 : Action INFO: Setting DC of HappyHour.Views.MainPage to HappyHour.ViewModels.MainPageViewModel.

12 : Action INFO: Attaching message handler HappyHour.ViewModels.MainPageViewModel to  
 HappyHour.Views.MainPage.

13 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for get\_DrinkName.

14 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for set\_DrinkName.

15 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for get\_CanAskForIt.

16 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for SockItToMe.

17 : ViewModelBinder INFO: Binding Convention Not Applied: Element AskForIt did not match a property.

To save space, I’ve elided the portion of each message that reports the date, time, and the name of the class method that is adding to the log:

1/16/2012 5:12:46 PM : : Cocktail.DefaultDebugLogger:LogWriter :

The line #10 tells us that Cocktail bound the *MainPage* to the *MainPageViewModel*. We know that already but it’s nice to see it confirmed; the fact that it is available in the log means we could write tests or runtime checks to detect *View* binding failures at runtime if we wished to do so.

The “*DC*” in “*Setting DC of HappyHour.MainPage*” is the *DataContext*. That’s important information. We want to know that the *MainPage* is bound to the expected data source, the *MainPageViewModel*.

The binding problems are apparent in the subsequent messages … if we know what we are looking for.

We can ignore most of the “Action Convention” messages. *Actions* are behaviors associated with UI triggers such as button clicks and mouse-overs. We don’t expect UI triggers to be bound to the ViewModel’s *DrinkName* or *CanAskForIt* properties so we aren’t perturbed by messages that say “*No actionable element for …*” with respect to these properties.

However, we should be concerned that no “*actionable element*” was found for *SockItToMe*. We expect that *MainPageViewModel* method to be called when the user clicks the button.

The Binding Convention message “*Element AskForIt did not match a* property” is the other side of that coin. It tells us that a UI element named “*AskForIt*” (the button) was not bound to a member of the *ViewModel*. When you program in the style we recommend, most of your UI element names (“*x:Name*” attribute values) should be bound to a *ViewModel* member. A named element that is not bound is worth investigating.

Try to keep the noise down by eliminating unnecessary element names. You don’t want to be distracted by elements that aren’t supposed to be bound. That’s why we removed the name “LayoutRoot” from the generated <grid> xaml.

We are also **missing some messages**. In general, the public members of a *ViewModel* are bound to a *View* element. We should have seen confirmation of a binding to *MainPageViewModel.****DrinkName***. We didn’t and that fact should lead us to search the *View*’s XAML for a control that was supposed to be bound to the *DrinkName* property (the *TextBox* as we know). Either the “x:Name” wasn’t specified (the culprit in this case) or the value doesn’t match the *ViewModel* property name.

Let’s undo the damage and see what the log says when the application is configured properly.

**Restore the *TextBox* *x:Name*** attribute in *MainPage*.

<TextBox x:Name="DrinkName" BorderThickness="1" BorderBrush="Blue" Margin="4" />

**Restore the *AskForIt*** method name in *MainPageViewModel*.

public void AskForIt()

**Build and run [F5]**

The application should work properly again. Enter one letter and click the button. When we check the Output window we see:

10 : ViewModelBinder INFO: Binding HappyHour.Views.MainPage and HappyHour.ViewModels.MainPageViewModel.

11 : Action INFO: Setting DC of HappyHour.Views.MainPage to HappyHour.ViewModels.MainPageViewModel.

12 : Action INFO: Attaching message handler HappyHour.ViewModels.MainPageViewModel to  
 HappyHour.Views.MainPage.

13 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for get\_DrinkName.

14 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for set\_DrinkName.

15 : ViewModelBinder INFO: Action Convention Not Applied: No actionable element for get\_CanAskForIt.

16 : ViewModelBinder INFO: Action Convention Applied: Action AskForIt on element AskForIt.

17 : ViewModelBinder INFO: Binding Convention Applied: Element DrinkName.

18 : ActionMessage INFO: Action: AskForIt availability update.

19 : ActionMessage INFO: Action: AskForIt availability update.

20 : ActionMessage INFO: Action: AskForIt availability update.

21 : ActionMessage INFO: Action: AskForIt availability update.

22 : ActionMessage INFO: Invoking Action: AskForIt.

Message #16 confirms that the *AskForIt* method is now bound to a UI element (the button) named “AskForIt”. Message #17, “*Binding Convention Applied*”, confirms that a UI element called “DrinkName” was bound to a matching property.

Messages #18 through #21 were triggered by updates to the *DrinkName* property which raises the *PropertyChanged* event on the *CanAskForIt* guard property. The messages both indicate that activity occurred and relate it to the “AskForIt” UI element. That means Cocktail picked up the fact that the *CanAskForIt* property governs the enabled state of the *AskForIt* button.

Finally, message #22 tells us that the “AskForIt” button invoked its associated action, which is to say, the button-click triggered the *ViewModel*’s *AskForIt* method and popped up a *MessageBox*.

# Last Call

We learned a little more about how the conventions work (and we expand upon convention binding below). Mistakes are always possible, especially when you use "magic strings" to specify ViewModel property names as you must when working in XAML; you're vulnerable in this respect whether you use conventional or explicit data binding.

Fortunately, Cocktail logs binding behavior to the Visual Studio Output window. Understanding the logs can help you detect and repair binding mistakes ... which could make you more comfortable with the convention binding "magic".

# Ask the Mixologist

This lesson is finished. Feel free to move on directly to the next one. This “Ask the Mixologist” section is an optional digression from the lesson’s main course to related points of interest.

## Cocktail or Caliburn?

Throughout this lesson we referred to **Cocktail** as the agent behind the convention-based binding. It would be more precise to say that **Caliburn Micro** is the agent. Caliburn Micro is a key component of the ensemble that is Cocktail ... one of several components. Rather than confuse you by calling out each one individually, we'll just say that "Cocktail is doing it." When we think it's important to identify the specific contributing technology, we'll be sure to do so.

## How did it bind MainPage to MainPageViewModel?

In Cocktail we usually start with a ViewModel and expect to find the corresponding View class by name. The stock naming convention anticipates most of the common English language pairings.

The most common convention expects the view name to end in “*View*”:

<*BaseName*>**ViewModel** => <*BaseName*>**View**

Examples:

* *Customer***ViewModel** => *Customer***View**
* *Account***ViewModel** => *Account***View**

But the convention also accommodates synonyms for “*View*” such as “*Page*”, “*Form*” and “*Screen*”:

<*BaseName*><***ViewSynonym***>**ViewModel** => <*BaseName*><***ViewSynonym***>

Examples:

* *Customer****Page*ViewModel** => *Customer****Page***
* *Customer****Form*ViewModel** => *Customer****Form***
* *Customer****Screen*ViewModel** => *Customer****Screen***

We’re using this view-synonym convention in our tutorial to match *Main****Page*ViewModel** to *Main****Page***.

View/ViewModel naming conventions are much richer than described here.

## What are the naming conventions?

Learn more about naming conventions and how to change them:

* “[All about Conventions](http://caliburnmicro.codeplex.com/wikipage?title=All%20About%20Conventions)”
* “[View/ViewModel Naming Conventions](http://caliburnmicro.codeplex.com/wikipage?title=View%2fViewModel%20Naming%20Conventions)”
* “[Using the NameTransformer](http://caliburnmicro.codeplex.com/wikipage?title=Using%20the%20NameTransformer)”
* “[Handling Custom Conventions](http://caliburnmicro.codeplex.com/wikipage?title=Handling%20Custom%20Conventions)”
* [A CodePlex discussion post](http://caliburnmicro.codeplex.com/discussions/258997) that reveals how View/ViewModel binding conventions were determined.

These resources cover some basic material but quickly go deep into details and options in a way that may seem overwhelming. Fortunately, you don’t need to know any more right now than what we’ve shown you. We will drill into the advance conventions in more leisurely fashion in future lessons.

## Can I specify some data bindings explicitly?

Absolutely! Implicit convention-based binding exists to simplify tedious repetitive binding tasks. Use it when you think Cocktail *should know* what you mean. But you can always take the wheel and drive yourself.

Here is a rewrite of the “DrinkName” *TextBox* with explicit data binding.

<TextBox Text="{Binding DrinkName,

                Mode=TwoWay,

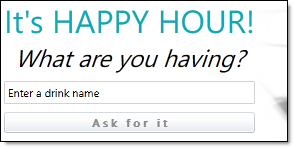
                NotifyOnValidationError=True,

                ValidatesOnNotifyDataErrors=True,

                TargetNullValue='Enter a drink name'}"

         Margin="0,8,0,8" />

We went the explicit route because we wanted to specify a value to display when the bound property (*DrinkName*) is null, an option not available with the stock conventions. Here’s what it looks like at runtime.



Notice that the button is not enabled. The *DrinkName* property is actually null; the binding is painting “*Enter a drink name*” into the *TextBox*; it’s not really there. You have to erase that text and type in a new value to enable the button and display the message.

The user experience is not good in our sample but we can imagine other circumstances in which the *TargetNullValue* binding property could be useful. We’re sure you’ll find reasons to bind a control in an unconventional manner. It’s good to know that you can and that your explicit bindings always trump the conventional implicit bindings.

## How is the *AskForIt* method bound to the button?

In many other MVVM frameworks, you’d have to create some form of “RelayCommand” in your ViewModel and bind to it explicitly in the View with attached properties. The “RelayCommand” in turn would delegate to the *CanAskForIt* and *AskForIt* members pretty much as we wrote them.

We had to write *CanAskForIt* and *AskForIt* … that’s the business logic that only we developers can know. But we don’t need any of the other ceremony in a Cocktail application. There’s nothing superfluous in the *ViewModel*; the XAML exhibits the same “x:Name” convention binding we use for data properties:

<Button x:Name="AskForIt" Content="Ask for it" Margin="0,0,0,4" />

Behind the scenes, Caliburn applies its conventions to bind the button to the guard property (*CanAskForIt*) and action method (*AskForIt*). The mechanism is called an Action and it’s much more powerful than what you see here. [Learn more about Actions](http://caliburnmicro.codeplex.com/wikipage?title=All%20About%20Actions) in the Caliburn documentation.

## Can I log my own messages?

Sure you can … with the Cocktail logging facilities.

**Add a call to Cocktail.LogFns.*DebugWriteLine*** in the *AskForIt* method.

public void AskForIt()

{

    Cocktail.LogFns.DebugWriteLine("Called AskForIt");

    MessageBox.Show(

        string.Format(CultureInfo.CurrentCulture,

        "One {0}, coming right up!", DrinkName)); // don't do this in real app

}

Run it and click on the button. You should see a line such as the following at the bottom of the Output window.

27 : ... HappyHour.ViewModels.MainPageViewModel:AskForIt: Called AskForIt

## Debug logs do not appear in Release builds

The logs are quite verbose. You probably don’t want all of that logging activity slowing down your application. Fortunately, the Cocktail convention logger only writes the logs in a Debug build. In Release builds, the convention binding log messages are ignored.

The *Cocktail.LogFns* methods that begin *Debug…* only apply to Debug builds; its *Trace…* methods are effective in all builds.

You can filter log writing and enable some or all logging in Release builds if you wish. Learn how in the Cocktail documentation; such customizations are out of scope for this tutorial.

Lesson 4: Data in a *ListBox*

Right now the “Happy Hour” application pops up a message box promisig to get us a drink. We’ll change that. We’re finally ready to put the (M)odel in MVVM by binding to an entity defined in its own model project.

The revised application will accept a drink name, create a new *DrinkOrder* entity with that name, add this new *DrinkOrder* to a *DrinkOrders* collection, and display that collection in a *ListBox*.

The **04-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

Here’s the ListBox version in action:



# Add a Model Project

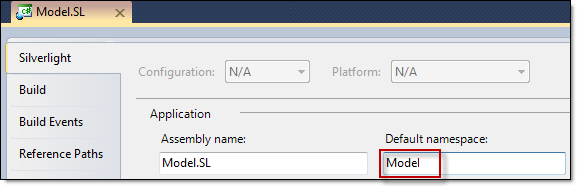
**File |Add | New Project | Silverlight Class Library | “*Model.SL*”**

The “.SL” suffix anticipates the non-Silverlight model project that awaits us in a future lesson.

**Pick “Silverlight 4”** or **“Silverlight 5”** depending upon which Silverlight version you’re building.

**Delete the template-generated *Class1***

**Change the project’s default namespace to “Model”** in the *Model.SL* project property sheet.



This is another anticipatory maneuver. The namespace of entities in this Silverlight project must be the same as the namespace of entities in the forthcoming non-Silverlight model project.

**Add | Class | “*DrinkOrder*”**

**Replace the contents** with the following definition of a *DrinkOrder* entity

namespace Model

{

    public class DrinkOrder

    {

        protected static int NextId = 1;

        public DrinkOrder()

        {

            Id = NextId++;

            DrinkName = "<new DrinkOrder>";

            Created = System.DateTime.UtcNow;

        }

        public int Id { get; private set; }

        public string DrinkName { get; set; }

        public System.DateTime Created { get; private set; }

    }

}

That's all will do in the *Model* project for now. Return to the *HappyHour* Silverlight application project.

# The *ViewModel* has *DrinkOrders*

**Add a *Model.SL* project reference to the *HappyHour* project**

**Open *MainPageViewModel***

**Add a *DrinkOrders* property** of type *BindableCollection<DrinkOrder>*

public BindableCollection<DrinkOrder> DrinkOrders {get; set;}

*BindableCollection<T>* is an enrichment of the .NET *ObservableCollection<T>*.

You’ll add two “using” statements in this process,

using Model;

using Caliburn.Micro;

Soon we’ll add a *ListBox* to the view and we’ll name it “DrinkOrders”, the same as the property name. A convention will kick in, binding the *ListBox.ItemsSource* to this *DrinkOrders* property automatically.

Of course the *ViewModel* itself betrays no awareness of this future; its job is to make the *DrinkOrders* available in a bindable form. Mission accomplished.

**Add a constructor to initialize *DrinkOrders***

 public MainPageViewModel()

 {

     DrinkOrders = new BindableCollection<DrinkOrder>();

 }

**Add a SelectedDrinkOrder property**

private DrinkOrder \_selectedDrinkOrder;

public DrinkOrder SelectedDrinkOrder

{

    get { return \_selectedDrinkOrder; }

    set {

        if (\_selectedDrinkOrder == value) return;

        \_selectedDrinkOrder = value;

        NotifyOfPropertyChange("SelectedDrinkOrder");

    }

}

We want to bind this property to the ListBox.SelectedItem so that the ViewModel hears when the user selects a drink in the ListBox.

We will reset the SelectedDrinkOrder each time we add a new drink. We want the ListBox to hear about that and adjust its selected item accordingly so we have to raise the PropertyChanged event when the SelectedDrinkOrder value changes.

We could write the binding by hand … but we don’t have to. The property name, SelectedDrinkOrder, adheres to a convention for Selector controls. If the Selector control's name is "Foos", the convention says that "SelectedFoo" (or "ActiveFoo" or "CurrentFoo") should be bound to the Selector control’s SelectedItem.  SelectedDrinkOrder fits the pattern for a ListBox named "DrinkOrders".

**Replace every “AskForIt” with “AddDrinkOrder” in the entire project** . Use Visual Studio's global search-and-replace.

We are re-purposing the *Button* to add new *DrinkOrder* objects to the *DrinkOrders* collection.

**Replace the body of *AddDrinkOrder*** (formerly *AskForIt*)

public void AddDrinkOrder()

{

    var drink = new DrinkOrder {DrinkName = DrinkName};

    DrinkOrders.Add(drink);

    SelectedDrinkOrder = drink;

}

We don't need AskForIt anymore but we do need an AddDrinkOrder method. Instead of launching a *MessageBox*, we create a new *DrinkOrder* with the user-entered drink name and add that *DrinkOrder* to the *DrinkOrders* collection. We’re also making the new drink order the “selected drink order” … a fact that should become apparent on screen.

The entire *MainPageViewModel* should look like this:

using System;

using Caliburn.Micro;

using Model;

namespace HappyHour

{

    [Export]

    public class MainPageViewModel : Screen

    {

        public MainPageViewModel()

        {

            DrinkOrders = new BindableCollection<DrinkOrder>();

        }

        public BindableCollection<DrinkOrder> DrinkOrders { get; set; }

        private string \_drinkName;

        public string DrinkName

        {

            get { return \_drinkName; }

            set

            {

                \_drinkName = value;

                NotifyOfPropertyChange("CanAddDrinkOrder");

            }

        }

        private DrinkOrder \_selectedDrinkOrder;

        public DrinkOrder SelectedDrinkOrder

        {

            get { return \_selectedDrinkOrder; }

            set {

                \_selectedDrinkOrder = value;

                NotifyOfPropertyChange("SelectedDrinkOrder");

            }

        }

        public bool CanAddDrinkOrder

        {

            get { return !String.IsNullOrWhiteSpace(DrinkName); }

        }

        public void AddDrinkOrder()

        {

            var drink = new DrinkOrder {DrinkName = DrinkName};

            DrinkOrders.Add(drink);

            SelectedDrinkOrder = drink;

        }

    }

}

# Present *DrinkOrders* in a *ListBox*

**Open *MainPage.xaml***

**Convert the *StackPanel* to a five row *Grid*.**

<Grid>

    <Grid.RowDefinitions>

        <RowDefinition Height="Auto" /> <!-- Title -->

        <RowDefinition Height="Auto" /> <!-- Question -->

        <RowDefinition Height="Auto" /> <!-- TextBox -->

        <RowDefinition Height="Auto" /> <!-- Button -->

        <RowDefinition Height="\*" />    <!-- ListBox -->

    </Grid.RowDefinitions>

    <TextBlock ... Grid.Row="0" />

    <TextBlock ... Grid.Row="1" />

    <TextBox x:Name="DrinkName" ... Grid.Row="2" />

    <Button  x:Name="AddDrinkOrder" ... Grid.Row="3" />

</Grid>

We can’t use the *StackPanel* anymore. We’re about to add a *ListBox* whose contents will grow potentially beyond the length of the browser window. We’ll want to scroll the *ListBox* when it does. Unfortunately, UI elements within a *StackPanel* can’t scroll. That’s by design … however inconvenient it may be for us. The contents of a *Grid* can scroll so we abandon the *StackPanel* in favor of the *Grid*.

**Add a *ListBox* named “DrinkOrders”** below the button; put it on the fifth row (row “4” counting from 0).

<ListBox x:Name="DrinkOrders" Grid.Row="4" />

By convention, the *ListBox.ItemsSource* will be bound to (and populated by) the *ViewModel*’s *DrinkOrders* property. When the user selects a row in the *ListBox*, the associated *DrinkOrder* entity becomes the *SelectedDrinkOrder*.

**Bind the *ItemTemplate* of the *ListBox* to “*DrinkOrderDataTemplate*”**:

<ListBox x:Name="DrinkOrders" Grid.Row="4"

         ItemTemplate="{StaticResource DrinkOrderDataTemplate}" />

The *ListBox* needs a *DrinkOrderDataTemplate*; time to write it.

## Write the *DrinkOrderDataTemplate*

We need a way to display the *DrinkOrders* in the *ListBox*. In this lesson, we are following the industry standard practice of displaying *ListBox* items in an ***ItemTemplate*** which is implemented as a *DataTemplate*.

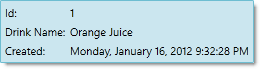
In the next lesson, we replace the *DataTemplate* with a *View* based on a *UserControl*.

The *MainPage* should stay focused on a consistently high level of display concerns, not the minutia of presenting a *DrinkOrder* in a ListBox. It should only have “main page” XAML. Therefore, we’ll put the *DataTemplate* in a separate *ResourceDictionary* and import that dictionary in the *MainPage*.

**Select the *Views* folder**

**Add | New Item | Silverlight Resource Dictionary | “*DrinkOrderDataTemplate.xaml*”**

The template will display a *DrinkOrder* on three lines like this:



**Define the *DrinkOrderDataTemplate*** as follows:

<ResourceDictionary

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

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

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

    <DataTemplate x:Key="DrinkOrderDataTemplate">

        <Grid cal:Bind.ModelWithoutContext="{Binding}" >

            <Grid.Resources>

                <!-- Re-implemented implicit styles for non-Controls -->

                <Style BasedOn="{StaticResource DefaultTextBlock}" TargetType="TextBlock" />

            </Grid.Resources>

            <Grid.RowDefinitions>

                <RowDefinition />

                <RowDefinition />

                <RowDefinition />

            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>

                <ColumnDefinition /> <!-- column for labels -->

                <ColumnDefinition /> <!-- column for values -->

            </Grid.ColumnDefinitions>

            <TextBlock Text="Id:"  Grid.Row="0" Grid.Column="0" Padding="2" />

            <TextBlock Text="Drink Name:" Grid.Row="1" Grid.Column="0" Padding="2" />

            <TextBlock Text="Created:"    Grid.Row="2" Grid.Column="0" Padding="2" />

            <TextBlock x:Name="Id"        Grid.Row="0" Grid.Column="1" Padding="2" />

            <TextBlock x:Name="DrinkName" Grid.Row="1" Grid.Column="1" Padding="2" />

            <TextBlock x:Name="Created"   Grid.Row="2" Grid.Column="1" Padding="2"

            Text="{Binding Created, StringFormat=\{0:F\}}" />

        </Grid>

    </DataTemplate>

</ResourceDictionary>

You could write this in a few lines of HTML. Sadly, we must endure in quiet rage the XAML madness that is necessary to achieve such paltry effect.

The *Grid* is typical for this kind of item display. It has three rows and two columns, the left column holding labels and the right holding the data values.

There is a Caliburn namespace at the top: *xmlns:cal="http://www.caliburnproject.org"*. We could have bound the controls explicitly. But we want to use the same convention based binding that we’ve enjoyed to this point. We have to reach out to Caliburn to get that.

The Grid element carries an attached property: *cal:Bind.ModelWithoutContext="{Binding}"*thattriggers convention-based binding of the template controls to the template’s runtime *DataContext* … which we know will be a *DrinkOrder*. Without this attached property, the view controls would just sit on the screen, unbound and inert.

The style resource for the *TextBlock* may surprise you; it surprised me. We defined an implicit style for the *TextBlock* in HappyHourStyles which is a *ResourceDictionary* in the App.xaml. That style should apply here automatically. It won’t for [reasons explained below](#_Why_import_the). We have to re-specify it somehow, perhaps with explicit styles or, as here, by re-implementing the TextBox implicit style in the context of the template.

Finally, notice that the “Created” *TextBlock* has an **explicit binding**

<TextBlock x:Name="Created" ...Text="{Binding Created, StringFormat=\{0:F\}}" />

We’ve decided to format the *Created* date in a more pleasing way with the *Binding*’s *StringFormat* attribute. That requires a custom binding which we must write explicitly.

An explicit binding always trumps conventional binding. Cocktail detects the explicit binding and gets out of the way. It’s now our responsibility to specify the binding completely and properly.

## Add the *DataTemplate* to the *MainPage*

**Open *MainPage.xaml***

**Add Resources** to the *MainPage.xaml*.

**Add the *DrinkOrderDataTemplate*** dictionary to the resources.

Here’s the resulting XAML:

<UserControl.Resources>

    <ResourceDictionary>

        <ResourceDictionary.MergedDictionaries>

           <ResourceDictionary Source="/HappyHour;component/Views/DrinkOrderDataTemplate.xaml"/>

        </ResourceDictionary.MergedDictionaries>

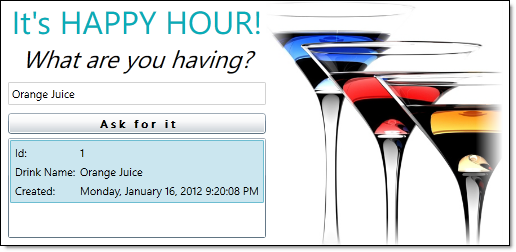
    </ResourceDictionary>

</UserControl.Resources>

You could merge this *ResourceDictionary* into the *App.xaml* instead of the *MainPage*; that would make sense if you expected to use the template(s) elsewhere in the application. Better to keep it a local resource until wider use is indicated.

**Build and run [F5]**

After adding a drink name and pressing the button you should see:



# Last Call

We created our first *Model* entity … a fake entity to be sure … and presented it in a *ListBox* in the standard way: with a *DataTemplate*.

The *DataTemplate* was bound by convention to the *DrinkOrder* when we added an attached property (*cal:Bind.ModelWithoutContext*) to the top element, the *Grid*.

We needed some ceremonial code to make it work. We re-implemented the *TextBlock* implicit style. We had to import the *DataTemplate* into the *MainPage* as a *DictionaryResource*.

We could have avoided that last step by defining the *DataTemplate* inline. I strongly dislike that approach for any but the most trivial template; the *MainPage* should be clean and focused on its purpose. Bloated XAML files with nested inline views are difficult to understand and maintain.

In the next lesson, we’ll try an alternative. We’ll replace the *DataTemplate* with a *View* class derived from *UserControl*.

# Ask the Mixologist

This lesson is finished. Feel free to move on directly to the next one. This “Ask the Mixologist” section is an optional digression from the lesson’s main course to related points of interest.

## Why create a Model project for DrinkOrder?

The *DrinkOrder* class is less than 20 lines of demo code. Why go to the trouble of creating a separate *Model* project? Couldn’t you simply add it to the main application project (*HappyHour*) and refactor to a Model project later … when you’re sure you need to?

Yes you could. But we suggest you follow our lead and isolate the model *right now*. Your model (not the tutorial model) is going to grow soon and, when it does, you’ll be glad it’s in its own project. Do it now and you avoid the circular dependencies between UI and Model components that can be difficult to disentangle when you discover them much later.

We know the agile axiom that urges you to “*do the simplest thing that could possibly work.*” That *seems* to argue against the Model project. Well, it doesn’t. Keeping the model in the application project can’t possibly work for long. The application project is dedicated to presentation; the model is separate, operating at a different layer. Distinct layers belong in their own assemblies.

The Model project is darned near the “simplest thing” you can do anyway. It takes less than a minute to create the Model project. After that, adding a class to the Model project is no more challenging than adding it to the application project. The Model project never slows you down and the clarity and maintenance benefits of a Model project are apparent as soon as you add a few more entities.

The “*keep it simple*” axiom exists to inhibit over-engineering, that expensive, time-consuming investment in complex designs for functionality to meet circumstances that *may* arise … *someday*.

Model growth is not an imaginary risk. It is a dead certainty that your model will exceed thirty entities within a cycle or two at most – not in this tutorial but definitely in your real project backed by real user stories. Experience teaches that a thirty entity model is sufficient to justify a separate project.

If the model doesn’t grow soon, you shouldn’t be reading this. You shouldn’t have bothered with a framework in the first place. You’re over-engineering the application just being here.

Seize the opportunity to do the “right thing” while the cost is low and the doing of it is all but inevitable.

## Should a property always notify the View?

The *DrinkOrders* property is implemented with a simple auto-property.

public BindableCollection<DrinkOrder> DrinkOrders { get; private set; }

Yet when we added the *SelectedDrinkOrder* property, we implemented it with a backing field and change notification.

private DrinkOrder \_selectedDrinkOrder;

public DrinkOrder SelectedDrinkOrder

{

   get { return \_selectedDrinkOrder; }

   set {

         \_selectedDrinkOrder = value;

         NotifyOfPropertyChange("SelectedDrinkOrder");

       }

}

Change notification alone forced us to write nine lines of boilerplate code. Why did we write those lines for *SelectedDrinkOrder* and not for *DrinkOrders*? Because *SelectedDrinkOrder* requires change notification and *DrinkOrders* does not.

The *SelectedDrinkOrder* changes every time the user clicks a button and adds a new drink.

public void AddDrinkOrder()

{

    var drink = new DrinkOrder {DrinkName = DrinkName};

    DrinkOrders.Add(drink);

    SelectedDrinkOrder = drink;

}

The *ViewModel* must tell the *View* about the new selected drink.

On the other hand, the *SelectedDrinkOrder* property never changes. The private setter is called exactly once, in the constructor, before the *View* can bind to it. The property always delivers the same collection instance even though the contents of the collection change.

If you want to play it safe … and you don’t mind a verbose *ViewModel* … you can implement every public property with a backing field and change notification. You’ll be consistent and maybe you have better things to think about than whether change notification is necessary. It never hurts.

I personally prefer the single-line economy of the auto-property. There is less code to wade through (one line versus nine) which may not matter in a small *ViewModel* but substantially obscures the purpose of a class with many properties.

A simple rule tells you when you need change notification and when you don’t:

**Add change notification if the *ViewModel* can set the property.**

Ultimately, this is a matter of personal and team preference.

## Why re-implement the *TextBlock* implicit style?

We define implicit control styles in the *HappyHourStyles.xaml* dictionary and we merge that dictionary into the *App.xaml* because want these styles to apply everywhere in our application.

They are applied automatically in *View* classes derived from *UserControl*. They are applied automatically within *DataTemplates* as well … but only to the elements that inherit from the *Control* class [as explained here](http://mosalem.blogspot.com/2010/07/silverlight-40-implicit-styles-not-so.html). Elements that are not *Control*s are not styled implicitly. They’re on the other side of something Microsoft calls “an encapsulation boundary.”

Templates are viewed as an encapsulation boundary when looking up an implicit style for an element which is not a subtype of Control.

*TextBlock* is not a *Control*; it’s a *FrameworkElement*. The implicit style for *TextBlock* does not flow through to the *TextBlocks* in our template. Microsoft says this is “by design.”

You are forgiven if the subtlety of the distinction is lost on you. Very few people know about this wrinkle; you could waste hours trying to figure out why your implicit styles weren't working. Of the remedies, we think re-implementing the implicit *TextBlock* style inside the template is the least onerous.

Lesson 5: Entity Views

We learn to use a *View* class instead of the *DataTemplate* to display entities in a *ListBox*. We update Cocktail conventions to treat entity classes as *ViewModel*s.

The **05-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

We saw in the previous lesson how to use a *DataTemplate* to display *DrinkOrder* entities in a *ListBox*. We were still able to use conventional *View/ViewModel* binding thanks to the ***Bind.ModelWithoutContext*** attached property.

It wasn’t pretty.

* We had to add that attached property to the *DrinkOrderDataTemplate*.
* We had to re-implement the *TextBlock* implicit style.
* We can’t visualize the *DrinkOrderDataTemplate* in the Designer.
* We can’t add a code-behind to a *DataTemplate* if we discover we need *View* behavior that is difficult to express in XAML.

We’ll try a different approach in this lesson. We’ll redefine the *View* of the *DrinkOrder* as a *UserControl* and marry that *View* to the *DrinkOrder* entity class as if that class were a *ViewModel*. We’ll change the stock conventions to recognize *View*s associated with entity classes in the *Model*.

# Define a *View* for the *DrinkOrder* entity

**Select the *Views* folder**

**Add | New Item | Silverlight User Control | “DrinkOrderEntityView”**

**Delete the *DrinkOrderEntityView.xaml.cs*** code-behind as we won’t be using it.

**Replace the empty <Grid> … </Grid>** with the following XAML plucked from the *DrinkOrderDataTemplate* that we wrote in the previous lesson:

<Grid>

    <Grid.RowDefinitions>

        <RowDefinition />

        <RowDefinition />

        <RowDefinition />

    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>

        <ColumnDefinition /> <!-- column for labels -->

        <ColumnDefinition /> <!-- column for values -->

    </Grid.ColumnDefinitions>

    <TextBlock Text="Id:"         Grid.Row="0" Grid.Column="0" Padding="2"/>

    <TextBlock Text="Drink Name:" Grid.Row="1" Grid.Column="0" Padding="2"/>

    <TextBlock Text="Created:"    Grid.Row="2" Grid.Column="0" Padding="2" />

    <TextBlock x:Name="Id"        Grid.Row="0" Grid.Column="1" Padding="2"/>

    <TextBlock x:Name="DrinkName" Grid.Row="1" Grid.Column="1" Padding="2"/>

    <TextBlock x:Name="Created"   Grid.Row="2" Grid.Column="1" Padding="2"

            Text="{Binding Created, StringFormat=\{0:F\}}" />

</Grid>

It’s much simpler now. We can see this *View* displayed appropriately on the Visual Studio Design canvas. We don’t have to re-implement the *TextBlock* implicit style because the ambient implicit style flows through to the *TextBlock*s in this template. We don’t need the attached property and we don’t need the *Caliburn* namespace.

**Delete *DrinkOrderDataTemplate.xaml*** from the *View* folder; we won’t need it again.

# Revise the MainPage.xaml

**Delete the *<UserControl.Resource>*** from near the top of the file; the item template is gone and we don’t need to define any other resources at this time.

**Delete mention of the *ItemTemplate* from the *ListBox*** which now reads:

<ListBox x:Name="DrinkOrders" Grid.Row="4" />

# First failed run

**Build and run [F5]**

Enter the letter “x” into the *TextBox*. The button lights up. Click the button; the *TextBox* clears and a *DrinkOrder* appears in the *ListBox* below the button … sort of.



# What went wrong?

I could tell you but it’s more important to learn how to diagnose a problem and how the system works.

First, let’s confirm that the application is populating the *ListBox* with *DrinkOrder* entities.

**Put a breakpoint on the *SelectedDrinkOrder* setter** in the *MainPageViewModel*

Enter another drink and press the button. Satisfy yourself that the *SelectedItem* is indeed a *DrinkOrder* entity. *ViewModel* plumbing is working fine.

Now review the log displayed in the Visual Studio Output window:

... ViewLocator WARN: View not found. Searched: .

... ViewModelBinder INFO: Binding System.Windows.Controls.TextBlock and Model.DrinkOrder.

... Action INFO: Setting DC of System.Windows.Controls.TextBlock to Model.DrinkOrder.

The warning tells us that Cocktail can’t find a *View* that it’s looking for after searching … well it doesn’t say where or what it searched for but experienced hands know that it searched the executing assembly for a *View* to go with the *DrinkOrder* entity.

**Apparently Cocktail thinks our *DrinkOrder* entity is a *ViewModel***! Why not? Any object can be a *ViewModel*.

The problem is that Cocktail lacks a convention that matches our *DrinkOrderEntityView* to a class in the *Model* namespace called “DrinkOrder”. Having failed to find a view, it substitutes the standard “Missing View” view which consists of a *TextBlock* prepared with the text “*Cannot find view for Model.DrinkOrder*.”

This is the same “Missing View” substitution that we saw in a previous lesson.

Finally, it binds the *DrinkOrder* to the *DataContext* of this artificial “Missing View” … which is pointless here but would make perfect sense if the “Missing View” were a real view.

## Create our own naming convention

Cocktail can’t anticipate every convention we dream up. Fortunately, the naming conventions are ours to amend. Customizing the naming conventions is an advanced feature but it’s not hard. We’d dearly like to see this work and drive home the point that *any* class can be a *ViewModel*. So let’s tweak the conventions … as if we knew what we were doing.

**Open the *AppBootstrapper*** class file.

**Override the base *Configure* method**

**Add a *NameTransformer* rule**

protected override void Configure()

{

    base.Configure();

    Caliburn.Micro.ViewLocator.NameTransformer

        // Map "Model" classes to views in "HappyHour.Views"

        .AddRule(@"^Model\.(\w\*)(?<basename>)", "HappyHour.Views.${1}Entity${basename}View");

}

Take a deep breath; naming rules are defined with “regular expressions” which are pretty hairy for all of us. We describe how this one works in detail in [commentary below](#_Explain_the_rule).

The short of it is that we’ve added a rule to map entity class names in the *Model* namespace to view class names in the *HappyHour.Views* namespace. There is a twist: to emphasize that this is no ordinary view, we want the particle “Entity” to appear between the basename and the suffix, that is, between “DrinkOrder” and “View”.

Thus, an entity class name such as “*Model.DrinkOrder*” will be transformed to a view class, “*HappyHour.Views.DrinkOrderEntityView*”.

We just added that view class to the project. At runtime, each time we click the button, Cocktail adds a new *DrinkOrder* instance to the collection, then pairs the *DrinkOrder* to a new *DrinkOrderEntityView* for display in the *ListBox*.

While we could override the naming conventions in other way, this is the easiest way to generalize the approach for model-to-view name transforms. We’ll have one rule that works for any entity in the model.

**Build and run [F5]**

Watch the Output window as you add each new *DrinkOrder*. You’ll see a sequence such as:

[34] ActionMessage INFO: Invoking Action: AddDrinkOrder.

[35] ViewModelBinder INFO: Binding HappyHour.Views.DrinkOrderEntityView and Model.DrinkOrder.

[36] Action INFO: Setting DC of HappyHour.Views.DrinkOrderEntityView to Model.DrinkOrder.

... Elided for brevity ...

[42] ViewModelBinder INFO: Binding Convention Applied: Element Id.

[43] ViewModelBinder INFO: Binding Convention Applied: Element DrinkName.

[44] ViewModelBinder INFO: Binding Convention Not Applied: Element Created has existing binding.

#34 represents the button click as before. But this time Message #35 confirms that Cocktail is “*Binding HappyHour.Views.DrinkOrderEntityView and Model.DrinkOrder*” as we intend.

Messages #42 and #43 tell us that the “Id” and “DrinkName” UI controls were data bound conventionally.

Message #44 reminds us that the UI element, “Created”, has an existing binding defined in the XAML (see above); that binding is preserved and the convention ignored.

# Last Call

The *DataTemplate* resource is the traditional way to display *ListBox* items … and we have nothing against that approach. You can use Cocktail’s conventional binding with that template just as you would with a full-blown *View* class. But *DataTemplate*s can be squirrelly in ways we discussed. When the item display becomes more complex, it may be better to render the entity with a *View* class rather than a data template.

You could write a *ViewModel* dedicated to supporting that *View* … but such a *ViewModel* would do no more than receive an entity when constructed and present it to the *View* for display. That strikes us as wasted effort, a case of MVVM purity getting in the way of productivity.

Instead, we modified the *View/ViewModel* binding conventions so that an entity can act as its own *ViewModel*. Just like that, the conventions “just work.”

# Ask the Mixologist

This lesson is finished. Feel free to move on directly to the next one. This “Ask the Mixologist” section is an optional digression from the lesson’s main course to related points of interest.

## How are Views displayed in the *ListBox*?

The log in the Visual Studio Output window is instructive.

[28] ConventionManager INFO: ItemTemplate applied to DrinkOrders.

It tells us that an “ItemTemplate” was applied to the “DrinkOrders” *ListBox*.

What “ItemTemplate”? We haven’t defined an “ItemTemplate” so Caliburn substitutes [a default *ItemTemplate*](http://caliburnmicro.codeplex.com/wikipage?title=All%20About%20Conventions) consisting of a *ContentControl*.

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

xmlns:cal="clr-namespace:Caliburn.Micro;assembly=Caliburn.Micro">

<ContentControl cal:View.Model="{Binding}"

VerticalContentAlignment="Stretch"

HorizontalContentAlignment="Stretch" />

</DataTemplate>

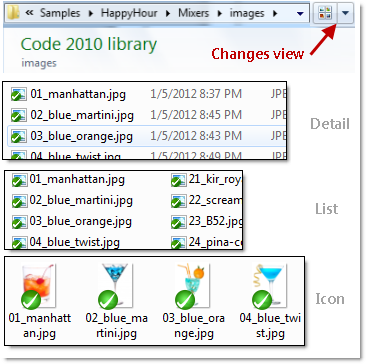
That *ContentControl* fills the available space with the *View* for the *ListBox* item which acts as the *ViewModel*. The “***cal:View.Model*”** attached property finds and constructs the proper *View* using the same *ViewLocator* mechanism that associated the *MainPage* with the *MainPageViewModel*.

That’s pretty clever. We put *ViewModels* in the *ListBox.ItemsSource* and let Cocktail compose the corresponding *Views* for display.

## What about multiple views for a *DrinkOrder*?

A *ViewModel* usually has only one *View*. When you see the *ViewModel* class name, you pretty much know the name of the one *View* that goes with it.

But you’ll find many examples of a ViewModel with multiple Views in the wild. Windows Explorer, for example, offers multiple views of the items in a directory:



You too can create an improved user experience with multiple perspectives over the same ViewModel. Here I’ll show how you can use conventions to display multiple views of an entity that is acting as a *ViewModel*.

Imagine you want to display a *DrinkOrder* in two different ways in two different contexts:



We already have the *DrinkOrderEntityView* on the left; let’s create a *DrinkOrderEntity.MiniView* to display the view on the right.

This example is included in the code accompanying this particular lesson only.

**Select the *Views* folder**

**Add | New Item | Silverlight User Control | “DrinkOrderEntity.MiniView.xaml”**

Take note of the period (.) between “DrinkOrderEntity” and “MiniView”. This view name follows the convention for alternative views:

Strip ‘View’ from the end of the normal view name, add a period (.), and add the suffix that distinguishes the alternate view.

We started with “HappyHour.Views.DrinkOrderEntityView”, stripped off “View”, added a period (.), and added “MiniView” to get “HappyHour.Views.DrinkOrderEntity.MiniView”.

Technically we’ve defined a *MiniView* class in the namespace *“HappyHour.Views.DrinkOrderEntity”*. Some might argue that we should have a corresponding, 3-deep folder structure, with a directory named “*HappyHour/Views/DrinkOrderEntity/*”. Suit yourself; that’s too picayune for my tastes.

The view itself is unremarkable, a single row grid with four columns as shown here.C:\Users\Ward\Documents\Visual Studio 2010\Projects\Cocktail\branches\DEV_Ward\Samples\HappyHour\Tutorial Images\05_DrinkOrderEntityMiniView.png

The corresponding XAML is:

<UserControl x:Class="HappyHour.Views.DrinkOrderEntity.MiniView"

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

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

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

    <Grid>

        <Grid.ColumnDefinitions>

            <ColumnDefinition />

            <ColumnDefinition />

            <ColumnDefinition />

            <ColumnDefinition />

        </Grid.ColumnDefinitions>

        <TextBlock Text="Id:"         Grid.Column="0" Padding="2"/>

        <TextBlock x:Name="Id"        Grid.Column="1" Padding="2"/>

        <TextBlock Text="Drink Name:" Grid.Column="2" Padding="2"/>

        <TextBlock x:Name="DrinkName" Grid.Column="3" Padding="2"/>

    </Grid>

</UserControl>

We’re using the same conventional binding as in *DrinkOrderEntityView* except we’re no longer displaying the *Created* property.

**Delete the** ***DrinkOrderEntity.MiniView.xaml.cs* code-behind** which isn’t needed.

**Open *MainPage.xaml***

Next we’ll tell the framework to use our newly-created *MiniView*.

[Recall](#_How_are_Views) that the framework binds the *ListBox.ItemTemplate* to a default *ItemTemplate* consisting of a *ContentControl* that hosts a *View*. We’ll substitute our own *ItemTemplate*

**Add the Caliburn namespace** so we can write some Caliburnattached properties.

<UserControl

...

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

... />

**Add the following XAML** just below the *<UserControl>* element

<UserControl.Resources>

    <DataTemplate x:Key="DrinkOrderItemTemplate" >

    <ContentControl

         cal:View.Context="MiniView"

cal:View.Model="{Binding}"

         HorizontalAlignment="Stretch"

VerticalAlignment="Stretch"/>

    </DataTemplate>

</UserControl.Resources>

This custom *ItemTemplate* is now a local resource of *MainPage*. It contains a *ContentControl* to hold an injected *View* just as the default template does.

As in the default template, the *cal:****View.Model*** attached property binds the *ContentControl.DataContext* to the *ViewModel* (the *DrinkOrder* in this example). That’s what triggered discovery and construction of the *DrinkOrderEntityView*.

But our *DrinkOrderItemTemplate* differs in one critical detail: the *cal:****View.Context*** attached property that specifies the alternate view’s suffix name, “MiniView”. Thanks to the *View.****Context*** , the application will discover and construct a *DrinkOrderEntity.MiniView* instead of a *DrinkOrderEntityView*.

Change the *ListBox* definition to use the *DrinkOrderItemTemplate* as its *ItemTemplate*.

<ListBox x:Name="DrinkOrders" ItemTemplate="{StaticResource DrinkOrderItemTemplate}" Grid.Row="4" />

This variation is present but commented out in the lesson’s code sample but you can revive it to see the alternative view in action.

**Build and run [F5]**

Try a few drink order names to confirm that you’re seeing the one-line *MiniView* of *DrinkOrders*.

## Explain the *NameTransformer* rule

We added our own *ViewModel-to-View* name transformation rule in the *Configure* method of the *AppBootstrapper*. The rule involved a pretty heavy [Regular Expression](http://www.regular-expressions.info/). Here I try to explain what that expression does.

Let me refresh your memory about the rule:

Caliburn.Micro.ViewLocator.NameTransformer

    // Map "Model" classes to views in "HappyHour.Views"

   .AddRule(@"^Model\.(\w\*)(?<basename>)", "HappyHour.Views.${1}Entity${basename}View");

The *NameTransformer* consists of a collection of rules that transform the name of a *ViewModel* class into the name of a *View* class. We’re adding a new rule that applies a search pattern (the first string) to the name of the *ViewModel* class. If the pattern matches, the *NameTransformer* applies the replacement (the second string) to produce candidate *View* names.

In our rule, the *ViewModel* will be an entity class in the “Model” namespace. The search pattern – *"****^Model\.(\w\*)* (?<basename>)** *"* – translates into English as:

*Match any class name that begins with ‘Model.’* (“Model dot”) *followed by any number of word characters followed by nothing.*

The parentheses in the pattern define a “capture group” that will contain the name of the entity class within the *Model* namespace.

The “***(?<basename>)***” is a placeholder capture group; it won’t capture a thing but it [needs to be there](http://caliburnmicro.codeplex.com/workitem/206).

The replacement pattern – “***HappyHour.Views.${1}Entity${basename}View***” – means:

*If you find a match, build a View class name consisting of ‘HappyHour.Views.’ followed by the first capture group followed by ‘EntityView’.” The “$(basename)” is always empty so ignore it.*

The first capture group is identified by the *${1}* token and refers to the entity class name discovered during the match. Here’s how it works for us:

* “*Model.DrinkOrder*” matches the pattern because it begins with “*Model.*”
* “*DrinkOrder*” is the matched text in group ${1}
* The replacement concatenates “*HappyHour.Views.*” + “*DrinkOrder*” + “*EntityView*”

Thus “*Model.DrinkOrder*” becomes “*HappyHour.Views.DrinkOrderEntityView*” as we intend.

Lesson 6: Talk to the View

Some View behaviors can be clumsy to stimulate with *View/ViewModel* binding. Although it offends the purists, we revive the *View* code-behind and teach the *ViewModel* to call into the *View* via an interface.

The **06-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

# Reset the view after each new drink order

Notice that, as you added new drink orders, the old value remains in the *TextBox*. Keep pressing the button and you place a new order for the same drink. That works for me; once I’ve found my drink, I tend to stay with it all night.

It may not work for the business. The specification says the *View* should prepare for a new choice after each order. We should clear the *TextBox* and set the focus there so it’s easy to make a new choice.

Should the *ViewModel* clear the *TextBox* and set focus? We don’t think so. The *ViewModel* shouldn’t know about *TextBoxes* and input focus. These are the details of interaction design decisions that are the proper concern of the *View*, not the *ViewModel*. The *ViewModel* should signal its intention to the *View* – it wants the *View* to “reset itself”. The *View* should respond to that signal in whatever visual and functional way the user experience designer deemed appropriate. This is a concrete example of the principle of “*separation of concerns*.”

## Supervising Controller

There is no easy way in XAML to set the *TextBox* focus at the *ViewModel*’s request.

I’m not saying you can’t do it. I’m saying it’s not easy. A simple thing like this should be easy.

An obvious approach would be to put the *TextBox* focus logic in a reset method in the *View*’s code-behind and let the *ViewModel* call that method as appropriate.

This approach violates a fundamental tenet of MVVM: the *ViewModel* should never have a reference to the *View*. We’re going to do it anyway.

We’ll do it in a way that preserves *separation of concerns*. We won’t call the method “*ClearTextBoxAndGiveItFocus*”; the ViewModel doesn’t need to know those details. We’ll call it “*ReadyForNewDrink*” because that’s the intent.

We’ll define the *ReadyForNewDrink* method in an interface that insulates the *ViewModel* from any particular concrete *View*.

Finally, we’ll write the *ViewModel* code such that it only calls *ReadyForNewDrink* if the *View* supports the interface; if the *View* doesn’t, no big deal.

When our ViewModel maintains a reference to the View and tells the View what to do through an interface, we’re using the [Supervising Controller](http://www.martinfowler.com/eaaDev/SupervisingPresenter.html) pattern. This is another in the family of [UI Architectures](http://martinfowler.com/eaaDev/uiArchs.html) that, like MVVM, help us maintain separation between the easily tested, non-visual presentation logic and the hard-to-test UI widgets on screen.

## Define the interface

**Select the *ViewModels* Folder**

**Add | Class | “IMainPage.cs”**

**Replace entire contents with this:**

namespace HappyHour

{

    public interface IMainPage

    {

        void ReadyForNewDrink();

    }

}

**Open *MainPageViewModel***

*MainPageViewModel* can’t call *ReadyForNewDrink* until it “becomes aware” of a *View* that implements it. Fortunately, *MainPageViewModel* inherits from *Screen* …

[Export]

public class MainPageViewModel : Screen

… and *Screen* implements the *IViewAware* interface as [described below](#_What_is_IViewAware?). What you need to know is that *Screen* calls its *OnViewAttached* method when the *View* becomes available. *MainPageViewModel* can override that method to gain access to the *view* … as shown here.

protected override void OnViewAttached(object view, object context)

{

    \_view = view as IMainPage;

}

private IMainPage \_view;

Casting the *View* as *IMainPage* makes its *ReadyForNewDrink* method accessible.

**Revise the *AddDrinkOrder*** **method** to ready the view after adding a new drink order to the collection.

public void AddDrinkOrder()

{

    var drink = new DrinkOrder {DrinkName = DrinkName};

    DrinkOrders.Add(drink);

    SelectedDrinkOrder = drink;

    ReadyForNewDrink();

}

private void ReadyForNewDrink()

{

    if (null != \_view) \_view.ReadyForNewDrink();

}

Notice that our *ViewModel* remains testable despite its dependence on a *View*. That dependence is limited to the *IMainPage* interface which is trivial to fake.

Of course we’ll have to remember to pass a fake view into the *MainPageViewModel* when we test it. We can do that by casting the tested *MainPageViewModel* to *IViewAware* and calling *AttachView* with the fake view during setup of *AddDrinkOrder* tests.

## Teach *MainPage* to get ready … in code-behind

Now that the *MainPageViewModel* is calling *IMainPage.ReadyForNewDrink*, we have to make *MainPage* implement that method … and it can only do so in code-behind.

There is no code-behind for the *MainPage* at the moment. We deleted the *MainPage.xaml.cs* file a few lessons ago because we didn’t need it. It is our strong preference to eliminate code and files that aren’t contributing. No problem; we can restore it now that we need it.

**Project | Add | Class | “MainPage.xaml.cs”**

**Replace entire contents with this:**

namespace HappyHour

{

    public partial class MainPage : IMainPage

    {

        void IMainPage.ReadyForNewDrink()

        {

            DrinkName.Text = string.Empty;

            DrinkName.Focus();

        }

    }

}

Our implementation resets the view for the next new drink order by clearing the *TextBox* and giving it focus. You could [restore the constructor](#_Shouldn’t_the_ShellView) that calls *InitializeComponent* if you wish. We decline to do so; we’ll let Cocktail call *InitializeComponent* for us.

**Rebuild and run [F5]**

Enter a drink name and click the button. Immediately after, you should see that the *TextBox* is clear, the button disabled, and focus is in the *TextBox* where it is ready for your next drink.

# Last Call

In this lesson, we found ourselves wanting a user experience that is difficult to accomplish by MVVM-style data binding. The easier approach is to write the desired behavior in the *View* code-behind and invoke it from the *ViewModel*.

We wrote a *View* interface (*IMainPage*) to abstract the *View* from the *ViewModel*. We made sure that interface member names expressed the intention (“*ReadyForNewDrink*”) rather than dictating the user interaction (“ClearTextBoxAndGiveItFocus”).

We saw that a *ViewModel* derived from the *Screen* class gains access to the *View* through the *OnViewAttached* method and thereby becomes capable of calling *View* members through the *View* interface.

This business of a *ViewModel* calling a *View* through an interface is an example of the [**Supervising Controller**](http://www.martinfowler.com/eaaDev/SupervisingPresenter.html) pattern which is an alternative to the more common **ViewModel** pattern.

We recommend that you stick with the **ViewModel** pattern when you can; it generally yields code that is cleaner and easier to maintain (no need for view interfaces).

But don’t be dogmatic about it. Use **Supervising Controller** when the *ViewModel* initiates a change in the *View* that isn’t easy to communicate through data binding.

# Ask the Mixologist

## Is *View* code-behind evil?

We are not opposed to code in the code-behind. We’re wary of it because code-behind is difficult to test and developers have a bad habit of hiding business logic there. But we don’t mind if the code is simple (no conditional logic) and confined to purely local matters of design and usability. Try not to make a habit of it … please.

## Shouldn’t the *MainPage* code-behind call *InitializeComponent*?

Certainly *something* has to call *InitializeComponent* before the view can appear on screen. That *something* is the Caliburn Micro view composition process which is why we were able to delete the code-behind in the first place.

If you feel more comfortable with a more traditional looking code-behind that has a constructor that calls *InitializeComponent*, go ahead and add it. There is no harm in calling it twice.

## Why *IMainPage* belongs to the *ViewModels*

You might think *IMainPage* belongs in the *Views* folder with a *HappyHour.Views* namespace because it prescribes functionality to be implemented by a *View*. In fact it belongs with the *ViewModels*.

Someday we might have a third assembly of interfaces. It could make sense to move it there. But that’s complexity we don’t need today. Right now we need to determine to which folder and namespace this interface belongs.

The unshakeable principle is this: ***Views* may depend upon *ViewModels* but *ViewModels* may not depend upon *Views*.** To understand why, let’s indulge a couple of thought experiments.

Suppose that *IMainPage* were defined in the *Views* namespace and folder. Then suppose we later discover a good reason to breakout the *ViewModels* into their own assembly.

Because *MainPageViewModel* depends upon *IMainPage*, the *ViewModels* assembly would have to depend upon the *Views* assembly. But we know that *MainPage* must implement *IMainPage* which means the *Views* assembly would have to depend upon the *ViewModels* assembly. We’d be trapped in a circular dependency – Views depends upon ViewModels which depends upon Views – and circular dependencies are disallowed in .NET.

Suppose we want to reuse our *ViewModels* to support multiple clients written in different XAML technologies such as Silverlight, WPF, Windows Phone and Windows 8. We couldn’t do that; the *MainPageViewModel* dependence on *IMainPage* in the Silverlight *HappyHour Views* assembly would pin the entire *ViewModels* assemblyto the Silverlight client.

These problems disappear when *IMainPage* is defined in *ViewModels*. The *Views* assembly must depend upon the *ViewModels* assembly. But there is no reciprocal dependency from *ViewModels* to *Views*. The Views in Silverlight, WPF, Windows Phone and Windows 8 could each depend on the same *ViewModels* assembly (assuming other aspects of the technologies permitted such dependence).

**Define *View* interfaces of the Supervising Controller pattern in the *ViewModels* namespace, not the *Views* namespace**.

## What is *IViewAware*?

The *Screen* class implements *IViewAware*, an interface that makes a *ViewModel* “aware” of the *View* to which it is attached. That “awareness” is well short of deep familiarity; such familiarity would break the separation we seek between the visual manifestation of the *View* and the non-visual view support that is the proper role of the *ViewModel*.

But there is no harm (and plenty of benefit) in knowing a few things about a *View* when it is abstracted behind this interface.

public object GetView(object context = null) { }

public void AttachView(object view, object context = null) { }

public event EventHandler<ViewAttachedEventArgs> ViewAttached;

1. ***GetView*** is called first, giving the ViewModel an opportunity to provide its companion View instance directly rather than rely on the framework to create it. A *ViewModel* could return a concrete cached *View*. If it returns *null*,that means the framework should create the *View*. In *HappyHour*, where the *MainPageViewModel* is only displayed once, the base class implementation returns null and a new *View* is created.
2. ***AttachView*** is called next with the concrete *View* to which this *ViewModel* is bound (either the one returned by *GetView* or the one created by the framework). *Screen* implements this method privately. *MainPageViewModel* is not involved but can learn about the view by overriding *OnViewAttached* … as we did in this lesson.
3. ***ViewAttached*** is the event raised after *AttachView* completes. It informs other interested components that this *ViewModel* has been bound to its *View* and is ready for business.   
   *MainPageViewModel* doesn’t bother listening for this event because it gets the same information from *OnViewAttached*. *OnViewAttached* is called before handlers of the *ViewAttached* event.

Lesson 7: Images & ValueConverters

TBD



The **07-HappyHour** tutorial folder holds the state of this solution at the end of the lesson.

How to manage images as resources

How to extend CM with ValueConverter conventions, for image controls in this example

Hint at handling images coming from the database

Deals only with displaying them

# Add HappyHour images as resources

**Delete *cocktail\_log\_big.png*** (we’ll re-link to it shortly)

**Add new “*images*”** folder to the ***assets*** folder

**Add | Existing Item |** …

**Browse to the Mixers\images** directory

**Select all image files** (Ctrl-A)

**Open dropdown on the Add** button and choose “Add as **Link**”. We’re linking to these files; you could add them if you prefer.

**Open MainPage.xaml**

**Add “*images*” to the *cocktail\_logo\_big.png* file path**

<Image Source="/HappyHour;component/assets/images/cocktail\_logo\_big.png" ...

# Add *ImageFilename* property to *DrinkOrder*

We’d like to display a pretty picture with each drink. We’ll extend the DrinkOrder entity with the filename of one of the pictures we just added to the project.

**Open DrinkOrder** in the Model project and add the following property

public string ImageFilename { get; set; }

We’re still working with dummy entities at this stage of development so let’s fake the way image filenames are assigned.

**Add | Class | *DrinkImageFilenames*** to the Model project

**Replace the class definition** with the following:

public static class DrinkImageFilenames

{

    public static string GetNameById(int id)

    {

        var filename = ImageFilenames[id % ImageFilenames.Length];

        if (string.IsNullOrEmpty(filename)) return filename;

        // return base path + filename

        // **ToDo: Get rid of this base path!**

        return "/HappyHour;component/assets/images/" + filename;

    }

    private static readonly string[] ImageFilenames = new []

        {

            null, // drink with no image filename

            string.Empty, // drink with empty image filename

            "01\_manhattan.jpg",

            "02\_blue\_martini.jpg",

"badname.xxx", // image name that doesn't exist

            "03\_blue\_orange.jpg",

            "04\_blue\_twist.jpg",

            "05\_cinnamon.jpg",

// elided for brevity ... actual code sample has the full list

            "35\_gin\_and\_tonic.jpg",

            "36\_midori-rickey.jpg",

            "37\_scotch\_and\_soda.jpg",

        };

};

The class defines an array of image file names for demonstration purposes; we deliberately made the first two filenames “blank” so we can account for the two forms of “missing image” cases.

The *GetNameById* method returns a filename plucked from the *ImageFileNames* array given a *DrinkOrder.Id*. It prefixes a (non-blank) filename with a base path defined as the location of the actual file in the *HappyHour* assembly resources.

**This is a terrible idea**. No model class should be aware of any UI details, least of all the base path of image resources. We’re going to make a point of fixing this shortly.

**Return to DrinkOrder.cs**

**Add the *ImageFileName* initialization** to the constructor

ImageFilename = DrinkImageFilenames.GetNameById(Id);

# Show images in the DrinkOrderEntityView

**Open *DrinkOrderEntityView*.xaml**

**Add a white background color to the grid** so the image blends in

<Grid Background="White">

**Add a new *<ColumnDefinition/>*** **to the front** of the grid; we’ll put the image in this first column.

<Grid.ColumnDefinitions>

    <ColumnDefinition /> <!-- column for drink image-->

    <ColumnDefinition /> <!-- column for labels -->

    <ColumnDefinition /> <!-- column for values -->

</Grid.ColumnDefinitions>

**Add a *TextBlock*** element to the top of the grid to hold the image as follows:

<TextBlock x:Name="ImageFilename" Grid.Row="0" Grid.RowSpan="3" Grid.Column="0"

Margin="0,0,10,0" Height="80"/>

Yes, I said *TextBlock*, not *Image* … for a reason. We will want to see the image. But first we should confirm that we can deliver the proper image file path.

**Shift the other two columns to the right** one column

<TextBlock Text="Id:"         Grid.Row="0" Grid.Column="1" Padding="2"/>

<TextBlock Text="Drink Name:" Grid.Row="1" Grid.Column="1" Padding="2"/>

<TextBlock Text="Created:"    Grid.Row="2" Grid.Column="1" Padding="2" />

<TextBlock x:Name="Id"        Grid.Row="0" Grid.Column="2" Padding="2"/>

<TextBlock x:Name="DrinkName" Grid.Row="1" Grid.Column="2" Padding="2"/>

<TextBlock x:Name="Created"   Grid.Row="2" Grid.Column="2" Padding="2"

   Text="{Binding Created, StringFormat=\{0:F\}}" />

**Build and run (F5)**

Add a few drink orders and you’ll see something like this in the *ListBox*:



This is the same convention-based *TextBox* binding we’ve seen before. Now we’re ready to see the picture.

**Substitute *Image* for *TextBox***

<Image x:Name="ImageFilename" Grid.Row="0" Grid.RowSpan="3" Grid.Column="0"

Margin="0,0,10,0" Height="80"/>

**Clean the project first; then build and run (F5)**

Enter a few drinks. The first drink won’t show an image because its *ImageFilename* is “blank”. But the second one should look like this:



Cocktail recognizes that you are binding an *Image* control to the *ImageFilename* and inserts its ***PathToImageSourceConverter*** into the binding pipeline. That converter returns an *ImageSource* object (a *BitmapImage* object) acquired from a resource location defined by the *ImageFilename* string.

# Customizing the custom convention

That’s good progress. But I’d like to have a placeholder image appear when the *ImageFilename* is “blank”. And remember that hack we left in the model project where we prefixed the filename with a base-path? We’re doing that in the Model which is the wrong place! I want to fix both problems.

**Add a new ConfigureImageConverter method** to the *AppBootstrapper* as follows

private static void ConfigureImageConverter()

{

    PathToImageSourceConverter.PathFilter =

        path => string.IsNullOrEmpty(path) ?

            null : "/HappyHour;component/assets/images/" + path.Trim();

    PathToImageSourceConverter.MissingImage =

        PathToImageSourceConverter.GetImageFromPath("missing\_drink.jpg");

}

**Call ConfigureImageConverter** in the *Configure* method

protected override void Configure()

{

// ...

    ConfigurePathToImageSourceConverter();

}

Let’s review the *ConfigureImageConverter* starting with the first statement which assigns a custom *PathFilter* method to the *PathToImageSourceConverter*.

By default the *PathToImageSourceConverter* assumes that the path string is a full-formed, relative or absolute *URI*. That’s fine for *DrinkOrder.ImageFileName* as it is currently … with that evil base-path. We really don’t want anything in the Model project to know about resource locations in the Silverlight project. Far better to store just the filename in the *DrinkOrder* and let the *ValueConverter* solder the base-path to the front of it.

The ***PathFilter*** was designed with that purpose in mind. It takes an input string, manipulates it as you wish, and outputs the resulting string. In our example, we prefix the input string with the base-path (after a little guard logic to deal with empty inputs). Note that specific knowledge of the base-path is code here – in the UI project – which is acceptable.

In the second statement, we use the core conversion method (which incorporates the *PathFilter* we just wrote) to set a path to a “MissingImage”. The converter substitutes this path when the *DrinkOrder.ImageFilename* is “blank”.

Now let’s go back and remove the evil base-path logic from the Model project.

**Open *Model.DrinkImageFilenames*** again

**Reduce *GetNameById*** to this

public static string GetNameById(int id)

{

    return ImageFilenames[id % ImageFilenames.Length];

}

**Build and run (F5)** and enter a few drinks.



The first drink order shows an empty glass, signifying that the drink image is missing. The second displays the matching drink image.

# Last Call

TBD