Contents

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

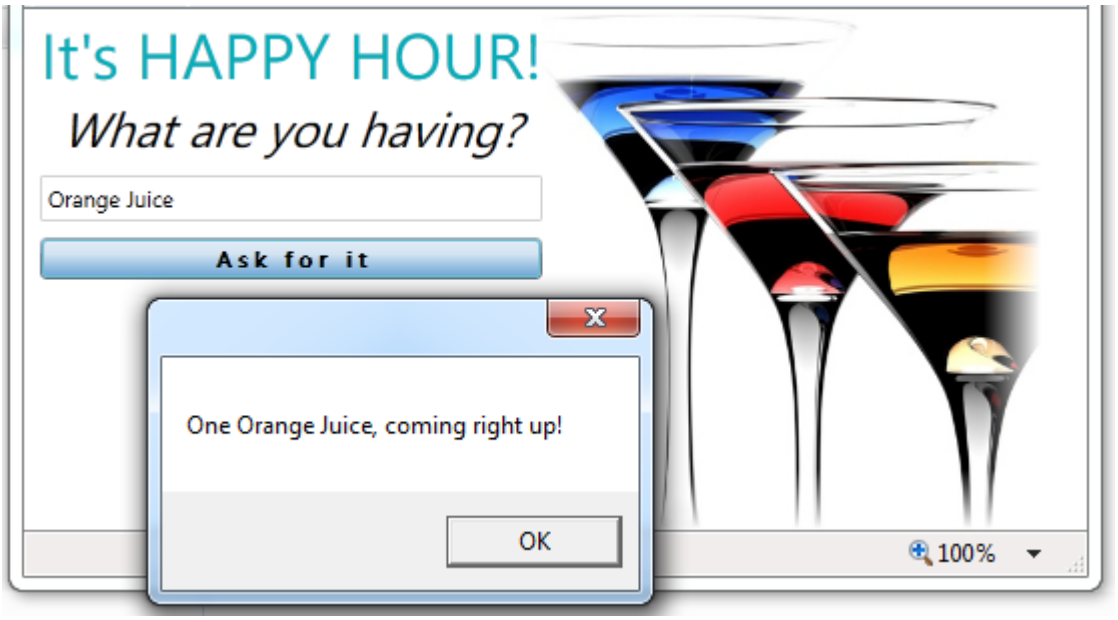
In this lesson, we'll look more closely at the conventions we've relied upon so far. Conventions are fine … as long as you follow them. If you inadvertently break with the conventions, your application can misbehave.

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/happyhour_logo.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 Punch's ability to connect the *MainPageViewModel* to the MainPage view and bind the View's controls to the ViewModel's members … all by means of commonplace naming conventions:

- *TextBox* **DrinkName** to *MainPageViewModel.DrinkName*.
- *Button* **AskForIt** to *MainPageViewModel.AskForIt* and *MainPageViewModel.CanAskForIt*.

We gave Punch 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 Punch 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 Punch 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 "*ViewModel*" 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.

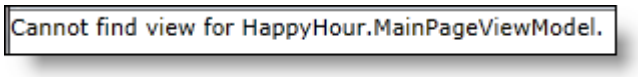
1. Open ***MainPage.xaml***.

2. Rename its class to **BadMainPage**.

```
<UserControl x:Class="HappyHour.BadMainPage"
```

3. **Rebuild and run** [F5].

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



Punch is looking for a View to go with the *MainPageViewModel*. It can't find one so Punch substitutes a default "Missing View" consisting of a single *TextBlock* with message text.

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

4. 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

Punch 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".

Punch associates those control names with the *DrinkName* property and the *AskForIt* method of the *MainPageViewModel* as you would expect. Punch 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.

1. 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" />
```

2. 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()
```

3. **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

1. **Run it again** [F5].

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

Punch 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 Punch 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.

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

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

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

```
public void AskForIt()
```

3. **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 Punch 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, Punch 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.

Punch or Caliburn?

Throughout this lesson we referred to **Punch** 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 Punch ... one of several components. Rather than confuse you by calling out each one individually, we'll just say that "Punch 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 Punch 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"
- "View/ViewModel Naming Conventions"
- "Using the NameTransformer"
- "Handling Custom Conventions"
- A CodePlex discussion post 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 Punch 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 x:Name="DrinkName" 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 *TargetNullValue* 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.

This 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 Punch 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 in the Caliburn documentation.

Can I log my own messages?

Sure you can … with the Punch 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.Text)); // 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 Punch logging class in use here is called the ***DefaultDebugLogger*** for a reason: it only writes the logs in a Debug build. In Release builds, the 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 Punch documentation; such customizations are out of scope for this tutorial.