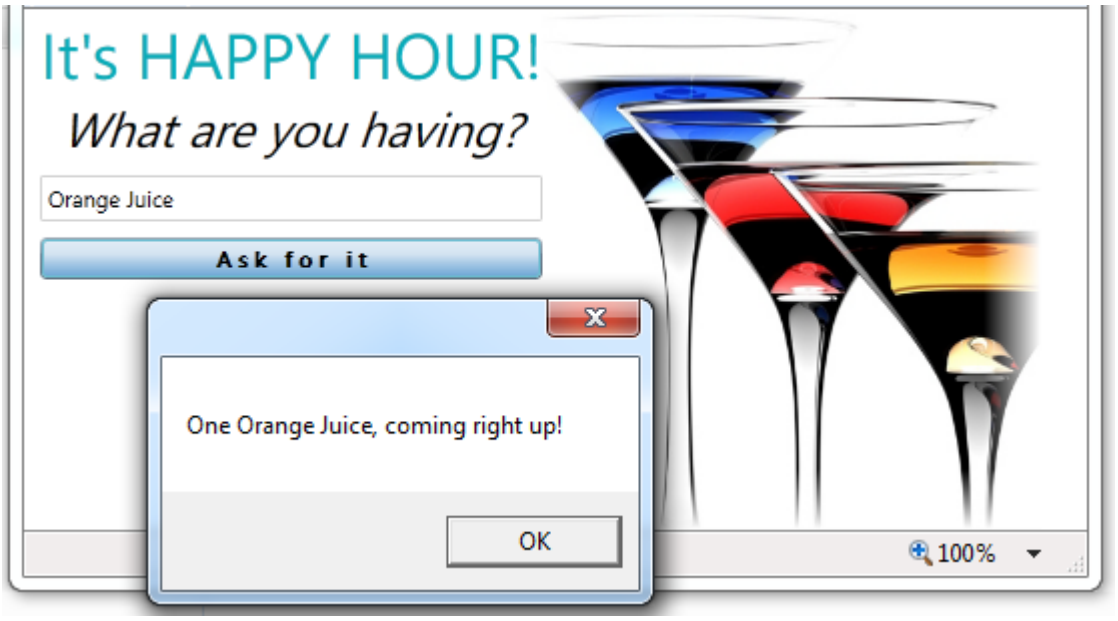
Contents

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 Punch 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.

Lesson 1 code review

The Lesson 1 solution is still in its pre-Punch 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.

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.

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

Add HappyHour project dependency to Punch

Adding the Punch dependency is done with NuGet. NuGet is a convenient and increasingly popular tool to manage third-party project dependencies. If you haven't done so already, install the NuGet extension to your Visual Studio.

1. **Right-click the HappyHour project, choose "Manage NuGet Packages…"**

2. Enter Punch in the search box. Make sure you are searching the "NuGet official package source".

3. Click "Install" next to "Punch.UI" and accept the license terms in order to download the Punch package and add the necessary references to the HappyHour project.

[Export] the *MainPageViewModel*

We're using MEF 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*:

```
using System.ComponentModel.Composition;
...
```

```
[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*.

Punch has its own startup code in a *CocktailMefBootstrapper* 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*.

1. **Add | New Item | Class | "AppBootstrapper"**

2. Inherit from ***Cocktail.CocktailMefBootstrapper<ViewModels.MainPageViewModel>***.

```
public class AppBootstrapper :
   Cocktail.CocktailMefBootstrapper<ViewModels.MainPageViewModel> { }
```

The *MainPageViewModel* type specification tells Punch 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.

3. 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>
```

You should rebuild the project at this point so Visual Studio can update the resources.

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.

4. 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 need now that we've introduced Punch.

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.

1. Open **MainPage.xaml**.

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

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

3. Remove the bindings from the Button … as shown here:

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

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/happyhour_logo.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 element names we choose follow naming conventions that enable Punch 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*; Punch 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. Punch 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. Punch 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

1. Open ***MainPageViewModel***.

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

3. Inherit from ***Screen*** instead of *INotifyPropertyChanged*.

```
[Export]
public class MainPageViewModel : Screen
```

Most of the ViewModels you write for a Punch 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*.

4. Delete the implementation of ***INotifyPropertyChanged***.

5. Delete everything having to do with **Command**.

6. Delete the ***MainPage* code-behind** (*MainPage.xaml.cs*).
We don't need it. We don't need the *AskForIt_Click* handler because Punch wires the button directly to the ViewModel's *AskForIt* method. We don't need to set the View's *DataContext* because Punch does that. We don't need to invoke the View's *InitializeComponent* method; Punch does that too.

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

8. **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.Text)); // 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. (For the curious, we'll discuss this binding by convention in the next lesson.)

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 Punch 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* it isn't performing any useful function. Punch (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, Talk 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**.