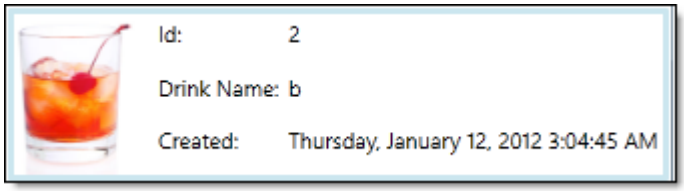
Contents

Here we cover techniques for **displaying images** that are stored as resources in the project, explaining how to access and configure a couple of essential image ***ValueConverters*** shipped in Cocktail. One of them could be used to display images stored in the database as byte arrays. The image-enriched *DrinkOrder* view looks like this:



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

Today's business applications make liberal use of images to inform and entertain. Some of these images are decorative elements of the Views. Some of these images are in the entity data that are stored in the database.
Whether fixed in the views or represented in entity data, you'll need some help displaying them on screen if you're using Data Binding. The Image element wants an *ImageSource* object; it is unlikely that your data source property delivers *ImageSource* objects. It more likely returns a string that translates to a URL of a resource … perhaps one shipped with the application or perhaps on the web.

It could also be a byte array of raw image data. We'll mention what to do about that later.

You'll probably need a *ValueConverter* to turn the string representation into an *ImageSource*.

You can find plenty of them on the web. But this is Punch where we try to make life a little easier for you. We ship two converters with Punch (one for string properties, one for byte arrays) and we wired both of them into the conventions. When you bind a string (or byte array) property to an Image element's Source property, we add the appropriate converter for you.

These converters are configurable. For example, Punch doesn't know exactly how to translate your application's image string into the *URI* it needs to construct an *ImageSource*. You'll have to provide the translator.
Punch doesn't know what image to display if your data bound image property is null, empty, or simply doesn't work. You can give the converter a "Missing Image" to use when this happens.

Let's put these ideas together in Happy Hour.

Add HappyHour images as resources

1. Delete **happyhour_logo.png** (we'll re-link to it shortly).

2. Add new **images** folder to the assets folder.

3. **Add | Existing Item | …**

4. Browse to the **Mixers\images** directory.

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

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

7. Open **MainPage.xaml**.

8. Insert **images/** into the **happyhour_logo.png** file path.

```
<Image Source="/HappyHour;component/assets/images/happyhour_logo.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.

1. 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.

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

3. 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 filenames 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.

4. Return to ***DrinkOrder.cs***.

5. Add the ***ImageFileName*** initialization to the constructor.

```
ImageFilename = DrinkImageFilenames.GetNameById(Id);
```

Show images in the DrinkOrderEntityView

1. Open **DrinkOrderEntityView.xaml**.

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

```
<Grid Background="White">
```

3. 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>
```

4. 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"/>
```

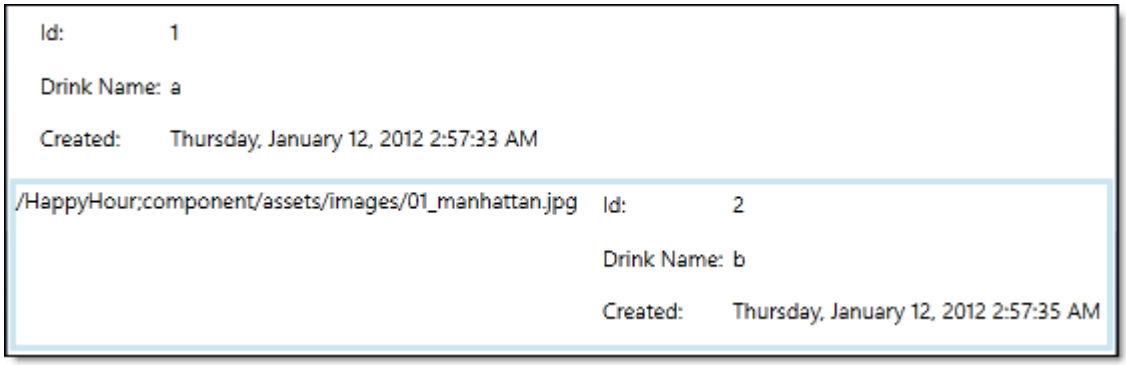
Yes, I said *TextBlock*, not *Image* … for a reason. We will want to see the image. But first we should confirm that we can deliver the proper image file path.

5. 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\}}" />
```

6. Build and run **(F5)**.

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



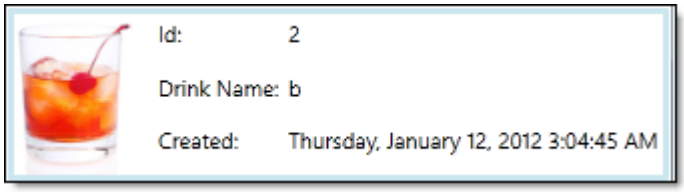
This is the same convention-based TextBox binding we've seen before. Now we're ready to see the picture.

7. Substitute ***Image*** for ***TextBlock***.

```
<Image x:Name="ImageFilename" Grid.Row="0" Grid.RowSpan="3" Grid.Column="0"
    Margin="0,0,10,0" Height="80"/>
```

8. 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:



Punch 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.

1. Add a new ***ConfigurePathToImageSourceConverter*** method to the ***AppBootstrapper*** as follows:

```
private static void ConfigurePathToImageSourceConverter()
```

```
{
  const string basepath = "/HappyHour;component/assets/images/";
   PathToImageSourceConverter.DefaultPathFilter =
      path => string.IsNullOrEmpty(path) ? path : basepath + path;
   PathToImageSourceConverter.DefaultMissingImage =
      PathToImageSourceConverter.GetImageFromPath(basepath + "missing_drink.jpg");
}
```

2. Call ***ConfigurePathToImageSourceConverter*** in the ***Configure*** method.

```
protected override void Configure()
{
  // ...
  ConfigurePathToImageSourceConverter();
}
```

Let's review the *ConfigurePathToImageSourceConverter* 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".

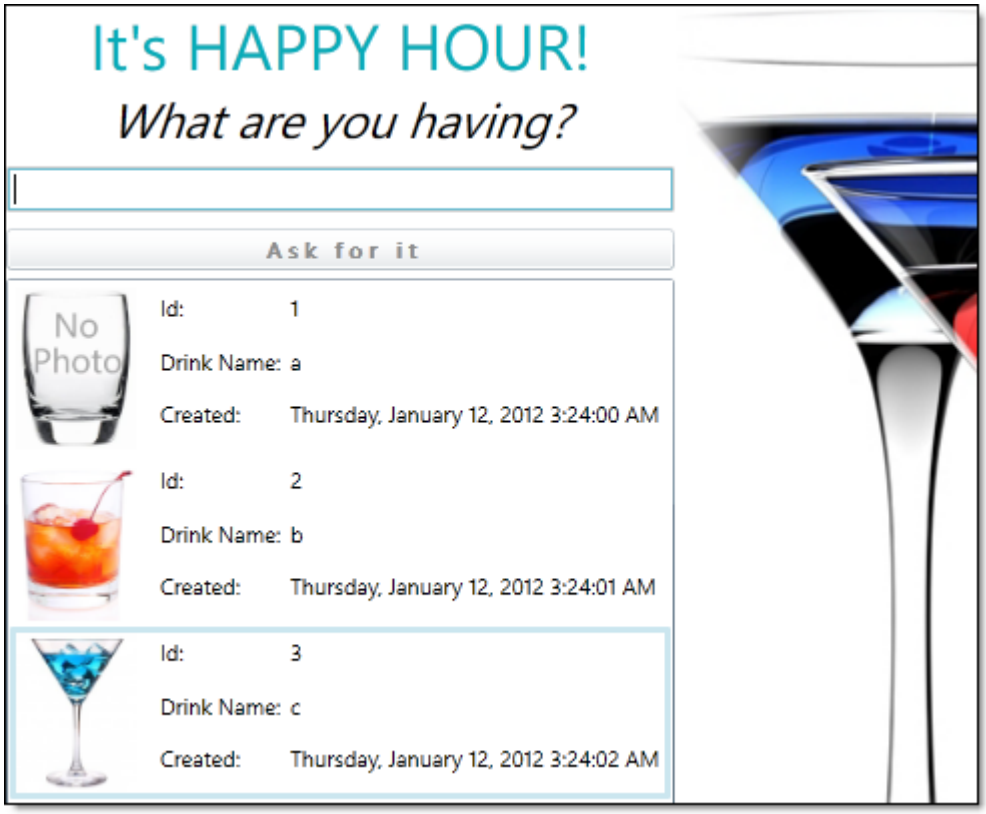
Now let's go back and remove the evil base-path logic from the Model project.

3. Open ***Model.DrinkImageFilenames*** again.

4. Reduce ***GetNameById*** to this:

```
public static string GetNameById(int id)
{
  return ImageFilenames[id % ImageFilenames.Length];
}
```

5. 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

That concludes this lesson in which we learned that Punch has built in *Image ValueConverters*. We learned how to configure the commonly used *PathToImageSourceConverter* to translate the image path in your data to a *URI* that references an application resource file. Your translator could as easily point to a web URL. We also saw how to provide a "missing image" that the converter will use when the converter can't convert the data bound image path string into an *ImageSource* object.

Ask the Mixologist

Feel free to move on to the next lesson … or tarry with the Mixologist to pick up a few advanced tips.

What about byte arrays?

The entity might store raw image data in the database record as a *byte array* instead of as a *URI* string. A Punch *Image ValueConverter* convention detects this property type and applies its *BinaryToImageSourceConverter*.

That converter is pretty simplistic in the current release: it lacks the *MissingImage* property and can only deal with the *byte arrays* that it recognizes as jpegs and pngs.

We'll improve it in future releases. Until then, you can write your own converter and register it with the other Punch *ValueConverters*.

How do I register my own ValueConverter?

A full discussion of this subject must await a future lesson. Meanwhile, take a look at the *ValueConverterConventionRegistry* class and its static *RegisterConvention* method in particular. You can register your own *ValueConverter* for a specified combination of control property and data bound property type. Your registered converter will trump a Punch converter for that [control property, data property] pair. The last registered converter always wins.

So go ahead, write a better byte array converter and register it in your application *bootstrapper* with code such as:

```
ValueConverterConventionRegistry.RegisterConvention(
    MyGreatByteArrayImageSourceConverter, Image.SourceProperty, typeof(byte[]));
```

And then please help everyone by contributing it to Punch :-).

How do I import an Image?

Good question [tap dances quickly]. We don't have a Punch for that one yet. You'll have to search the web for a user experience and import component that suits your application.