A Better Way to Data Bind Enums in WPF

Have you needed to data bind enums in WPF? Well, if you write WPF applications, you bet your ass you have! In this post, I would like to show you how I like to deal with enums in WPF. Fo this post, I will be using the following enum to bind to.

```
public enum Vehicle
{
    Horrible,
    Bad,
    SoSo,
    Good,
    Better,
    Best
}
```

The WPF Way

Normally if you were going to data bind a control to your enum you would need to go through a number of steps. First you need to add a XAML namespace for your local enum type and to System in MSCorLib.

```
xmlns:local="clr-namespace:BindingEnums"
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

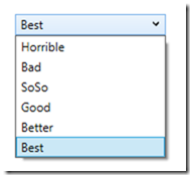
Then you would to create an ObjectDataProvider as a resource. Give it an x:Key so you can use it in your code, set the MethodName to "GetValues" which exists on the enum data type, and then set the ObjectType to that of the enum type. But wait, you're not done yet. Next you need to set the ObjectDataProvider.MethodParameters to that of your enum type. So you will end up with something like this.

```
<Window.Resources>
    <ObjectDataProvider x:Key="dataFromEnum" MethodName="GetValues"
                        ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="local:Status"/>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
```

Now, you you have something you can data bind to. For example, if I wanted to data bind a ComboBox to this enum, I would have to set it's ItemsSource to a new Binding, and set the source to that of our enum ObjectDataProvider we created above. Like so:

```
<Grid>
    <ComboBox HorizontalAlignment="Center" VerticalAlignment="Center" MinWidth="150"
              ItemsSource="{Binding Source={StaticResource dataFromEnum}}"/>
</Grid>
```

And viola! Our enum is data bound.



This isn't bad. It works just fine, but it requires a lot more code than I would like to have to write over and over. I would rather write the code once, and just change the enum type so I can reuse all that logic all over my application. With the above approach we have to create a new ObjectDataProvider for each enum we have, which just means more code to maintain, and more chances of writing broken XAML. Especially since you are more likely to copy and paste a previous ObjectDataProvider you created and just change some parameters. Hence, it's more error prone.

The Better Way

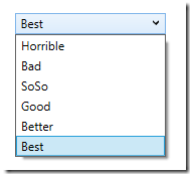
Let's look at how we can use features of WPF to improve the usage and readability of our code when data binding to enums. First, I want to encapsulate all that logic for creating a bindable list of enum values without the need of an ObjectDataProvider. I also want to eliminate the need to create a Resource that must be defined in order to be used in my XAML. Ideally I would just do everything inline like I would with a normal object binding. To accomplish this, I am going to enlist the help of a custom MarkupExtension. This extension will simply take an enum Type and then create a bindable list of enum values for my control. It's actually quite simple. Take a look:

```
public class EnumBindingSourceExtension : MarkupExtension
{
    private Type _enumType;
    public Type EnumType
    {
        get { return this._enumType; }
        set
        {
            if (value != this._enumType)
            {
                if (null != value)
                {
                    Type enumType = Nullable.GetUnderlyingType(value) ?? value;

                    if (!enumType.IsEnum)
                        throw new ArgumentException("Type must be for an Enum.");
                }

                this._enumType = value;
            }
        }
    }

    public EnumBindingSourceExtension() { }

    public EnumBindingSourceExtension(Type enumType)
    {
        this.EnumType = enumType;
    }

    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        if (null == this._enumType)
            throw new InvalidOperationException("The EnumType must be specified.");

        Type actualEnumType = Nullable.GetUnderlyingType(this._enumType) ?? this._enumType;
        Array enumValues = Enum.GetValues(actualEnumType);

        if (actualEnumType == this._enumType)
            return enumValues;

        Array tempArray = Array.CreateInstance(actualEnumType, enumValues.Length + 1);
        enumValues.CopyTo(tempArray, 1);
        return tempArray;
    }
}
```

Now we can simplify our XAML and simply create an enum binding like this:

```
<Grid>
    <ComboBox HorizontalAlignment="Center" VerticalAlignment="Center" MinWidth="150"
              ItemsSource="{Binding Source={local:EnumBindingSource {x:Type local:Status}}}"/>
</Grid>
```

Same result, but without the need for an ObjectDataProvider.



Now that's much better!

Adding Description Support

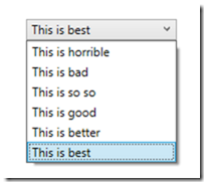
Now that we have our enum binding working perfectly without the need for that silly ObjectDataProvider, we can improve our enum experience. It is very common to have a enum to represent the values, while at the same time, give the enum value a description so that it is more readable to your user. For example; let's say you have an enum will all the state abbreviations (ID, TX, AZ, etc.), but you would much rather display the full state name (Texas, Idaho, Arizona, etc.). Wouldn't it be nice to have this ability? Well, luckily for us, this is easy to. We will just need to use a simple TypeConverter that we can attribute our enum with. Check it out:

```
public class EnumDescriptionTypeConverter : EnumConverter
{
    public EnumDescriptionTypeConverter(Type type)
        : base(type)
    {
    }

    public override object ConvertTo(ITypeDescriptorContext context, System.Globalization.CultureInfo culture, object value, Type destinationType)
    {
        if (destinationType == typeof(string))
        {
            if (value != null)
            {
                FieldInfo fi = value.GetType().GetField(value.ToString());
                if (fi != null)
                {
                    var attributes = (DescriptionAttribute[])fi.GetCustomAttributes(typeof(DescriptionAttribute), false);
                    return ((attributes.Length > 0) && (!String.IsNullOrEmpty(attributes[0].Description))) ? attributes[0].Description : value.ToString();
                }
            }

            return string.Empty;
        }

        return base.ConvertTo(context, culture, value, destinationType);
    }
}
```

Now add the attribute and add some descriptions to our enum:

```
[TypeConverter(typeof(EnumDescriptionTypeConverter))]
public enum Status
{
    [Description("This is horrible")]
    Horrible,
    [Description("This is bad")]
    Bad,
    [Description("This is so so")]
    SoSo,
    [Description("This is good")]
    Good,
    [Description("This is better")]
    Better,
    [Description("This is best")]
    Best
}
```

BAM! We now have our descriptions display to our users, but the actual values remain intact.



That does it for this post. Hopefully you will find this useful, and maybe even use this approach in your WPF applications. Be sure to check out the source code, and start playing with it. As always, feel free contact me on my blog, connect with me on Twitter (@brianlagunas), or leave a comment below for any questions or comments you may have.