Cardinal WPF Guide (v1.0)

Contents

Revisions

- v1.0 - Initial release.

MVVM Library

The MVVM Library provides a few classes to easily use the MVVM design pattern when using WPF. This section provides a simple overview of its function. In the WPF Style Library below we'll run through implementing the library.

To see an example application of these classes, there is a project named `MVVMLibraryExample` in source control.

PropertyChangedBase

INotifyPropertyChanged

This class should be the base class for all ViewModels created in your application. This base class provides an implementation of

INotifyPropertyChanged. To update the binding on your property using this Base class call the following code.

```
class MainWindowViewModel : PropertyChangedBase
{
    private string _content;
    public string Content
    {
        get { return _content; }
        set
        {
            _content = value;
            OnPropertyChanged();                // OR
            OnPropertyChanged(() => Content);   // OR
            OnPropertyChanged("Content");
        }
    }
}
```

There are three ways to implement the `OnPropertyChanged` function. Implicit name (using CallerMemberFunction), function passing name or string passing.

IDataErrorInfo

This base class also implements the IDataErrorInfo interface to allow for data validation. To check your data (properties only) are valid you first have to register the property in the constructor of the ViewModel. This data validation is mostly useful for TextBox or PasswordBox.

You can register this by passing in the property name, and a function that will be run whenever this property is updated. The function should return an error string if the property is invalid, or return null when it is valid.

```
public MainWindowViewModel()
{
    AddDataValidationForProperty(nameof(Content), ValidateContent);
}

private string ValidateContent()
{
    if (Content.Length < 6)
        return "Content must be at least 6 characters long";

    else
        return null;
}
```

This implementation provides a couple of other methods and properties you can make use of.

```
// Should the properties be validated every time a change is made, or manually.
bool AutoDataValidation

// Are all data properties valid
bool DataValidationOK

// Add data validation for property
void AddDataValidationForProperty(string property, Func<string> validationFunction)

// Remove data validation for property
void RemoveDataValidationForProperty(string property)

// Manually validate all properties that are being validated.
// This always works, but is supposed to be used along with AutoDataValidation = false
void ValidateData()

// This is used mostly for the UI to clear all error popups.
// (see WPF Style Library for implementation details)
void ClearValidationErrors()

// Gets the number of properties in error
int GetDataValidationErrorCount()

// Check if a particular property is in error
```

bool IsDataValidForProperty(string property)

RelayCommand

RelayCommand provides an implementation of the ICommand interface. It lets you easily get commands from the View to the ViewModel using the MVVM pattern.

To use it, first bind a command in the View, like this

```
<Button Command="{Binding ClickCommand}" />
```

Then create the command in the ViewModel using the following code. ClickCommand is the name of the `ICommand` object, and the method `ClickExec` is called when the button is clicked. `CanClickExec` returns true when the button command can be executed (it will disable the button when returning false).

```
public ICommand ClickCommand { get { return new RelayCommand(ClickExec, CanClickExec); } }

private bool CanClickExec()
{
    throw new NotImplementedException();
}

private void ClickExec()
{
    throw new NotImplementedException();
}
```

It is also possible to bind a command with a `CommandParameter` using the relay command like this. Here the `RelayCommand` is of type string, as that is the type of the command parameter.

```
<Button Command="{Binding ClickCommand}"
        CommandParameter="{Binding ElementName=MyTextBox, Path=Text}"/>



public ICommand ClickCommand { get { return new RelayCommand<string>(ClickExec, CanClickExec); } }

private bool CanClickExec(string textContent)
{
    throw new NotImplementedException();
}

private void ClickExec(string textContent)
{
    throw new NotImplementedException();
}
```

Messenger

The messenger class allows you to pass messages back and forth between ViewModels and even the main window without having hard references to each other. This can be very useful in MVVM when needing to bubble a message up or down depending on your architecture.

To implement a messenger, you need to create a Singleton of it. Open up the `App.xaml.cs` file and copy in the following code

```
internal static Messenger Messenger
{
    get { return _messenger; }
}

readonly static Messenger _messenger = new Messenger();

public const string MESSAGE_TEST = "Test";
```

For each message, we need a string name for the message, in this example, that string name is stored in App.xaml.cs alongside the messages to make it easier to access.

To send the message, use the following code. `message` can be of any type.

```
App.Messenger.NotifyColleagues(App.MESSAGE_TEST, message);
```

To receive the message, do the following

- Register for the message in the constructor for the view model.
- The register call must have a type specified that matches the type of `message` above.
- Use the same message name as when notifying above.
- Pass in the function that will be called when the message is notified. Again, this must match the type of the registered call.

```
public ViewModel1()
{
    App.Messenger.Register<string>(App.MESSAGE_TEST, UpdateMessage);
}

private void UpdateMessage(string message)
{
    MessageBox.Show(message);
}
```

WPF Style Library

This guide will step you through setting up the new Cardinal WPF Library in your application.

To see an example application of this style library there is a project named `WPFStylesLibraryExample` in source control.

Quick Start

Locate the `WPFStyleLibrary` and `MVVMLibrary` libraries in source control and add them both as references in your project.

Also add a reference to `System.Windows.Interactivity` (version 4.5).

NOTE: The project must be built in the latest version of .NET framework (4.6). If your new project is not at this level you will see a yellow triangle over the WPFStylesLibrary reference.

Open the application file `App.xaml` and add the following references -

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Controls.xaml" />
        <ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Icons.xaml" />
        <ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Fonts.xaml" />
        <ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Accent.xaml" />
        <ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/LightTheme.xaml" />
    </ResourceDictionary.MergedDictionaries>

 <!--  Add any other references here  -->
</ResourceDictionary>
```

Override the main window in your application (the entry point of your app). This will add the new features available in the library.

Open `MainWindow.xaml` and modify the following

- Add a reference to the library namespace `xmlns:controls="clr-namespace:WPFStylesLibrary.Controls;assembly=WPFStylesLibrary"`
- Change the main control from `<Window></Window>` to `<controls:SFWindow></controls:SFWindow>`
- Open `MainWindow.xaml.cs` and change the line `public partial class MainWindow : Window` to `public partial class MainWindow : SFWindow`

When you're all done with these changes, you should have the following content

MainWindow.xaml

```
<controls:SFWindow x:Class="Test.MainWindow"
                   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                   xmlns:controls="clr-namespace:WPFStylesLibrary.Controls;assembly=WPFStylesLibrary"
                   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
                   xmlns:local="clr-namespace:Test"
                   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
                   Title="MainWindow"
                   Width="525"
                   Height="350"
                   mc:Ignorable="d">
    <Grid />
</controls:SFWindow>
```

MainWindow.xaml.cs

```
using WPFStylesLibrary.Controls;

namespace __NAMESPACE__
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : SFWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

At this point we can confirm the references are all working correctly by running the solution. It should look like this --

Add the Messenger object into your `App.xaml.cs` file if you intend to do any messaging between ViewModels (you probably will, don't fight it). For more information about this, see Messenger / MVVM Library.

```
internal static Messenger Messenger
{
    get { return _messenger; }
}

readonly static Messenger _messenger = new Messenger();
```

Main Window ViewModel

Now all the references are made to the application, we need to create our first ViewModel. Following the MVVM Pattern, we need a corresponding ViewModel for every View (Window or UserControl) we have in the application. Convention dictates that for ever View name **MainWindow**.xaml we will have a ViewModel named **MainWindow**ViewModel.cs

All of the view models made in the application should inherit from PropertyChangedBase base class to give use of the extra functionality designed into this library.

Two of these new features are Notifications and Dialogs. To get these working, add the objects for `DialogHandler` and `NotificationHandler` into your `MainWindowViewModel.cs`. When you're done your class should look like this

```
using MVVMLibrary;
using WPFStylesLibrary.Dialogs;
using WPFStylesLibrary.Notifications;

namespace __NAMESPACE__
{
    class MainWindowViewModel : PropertyChangedBase
    {
        public DialogHandler DialogHandler { get; set; }
        public NotificationHandler NotificationHandler { get; set; }

        public MainWindowViewModel()
        {
            DialogHandler = new DialogHandler(App.Messenger);
            NotificationHandler = new NotificationHandler(App.Messenger);
        }
    }
}
```

`DialogHandler` and `NotificationHandler` both work similarly (and will be explained in their own sections) but essentially allow you to display Dialogs or Notifications from anywhere within your application (thanks to the reference to `App.Messenger`.

Now add the new objects as bindings in `MainWindow.xaml` under the main `<controls:SFWindow></controls:SFWindow>` section.

```
Dialog="{Binding DialogHandler.Dialog}"
Notifications="{Binding NotificationHandler.Notifications}"
```

Finally, set the `DataContext` of the MainWindow to make bindings possible. Setting the `DataContext` here will allow all bindings to work between the View and ViewModel. When you're complete, you should have this for your `MainWindow.xaml.cs`.

```
<controls:SFWindow x:Class="Test.MainWindow"
                   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                   xmlns:controls="clr-namespace:WPFStylesLibrary.Controls;assembly=WPFStylesLibrary"
                   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
                   xmlns:local="clr-namespace:Test"
                   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
                   Title="MainWindow"
                   Width="525"
                   Height="350"
                   Dialog="{Binding DialogHandler.Dialog}"
                   Notifications="{Binding NotificationHandler.Notifications}"
                   mc:Ignorable="d">

    <controls:SFWindow.DataContext>
        <local:MainWindowViewModel />
    </controls:SFWindow.DataContext>

    <Grid />
</controls:SFWindow>
```

Dialogs

To show Dialogs you first must follow the steps in the Quick Start and Main Window ViewModel sections to add the `Messenger` and `DialogHandler`.

All dialogs built into this Library are designed to be Modal, that is they must be addressed before the user can return to the application.

The Dialogs can be called from any ViewModel. This is done by sending a message to the Main Window of your application (in this case `MainWindow.xaml`) to display a selected `UserControl` and wait for response. This `UserControl` can be any View as long as it has an associated ViewModel that implements IDialog. See Custom Dialogs for information on how to create your own Dialog.

To display a dialog from any ViewModel use the following code. This is the long form of showing a dialog. See the standard dialogs section for the easy to use versions of this.

```
var newDialog = new DialogOKCancelViewModel("Are you sure you want to delete this super important thing?");
App.Messenger.NotifyColleagues(DialogMessage.SHOW_DIALOG, newDialog);

await newDialog.WaitForDialogClose();
var result = newDialog.Response;
```

This snippet creates the DialogViewModel and then passes it to the `MainWindow` using the `App.Messenger`. It then waits for a response from the user and sits on the `await` line until the dialog is closed. This will not block the UI.

Standard Dialogs

Just like with WinForms, this library provides a set of dialogs for the user to display in standard situations. The OK/Cancel and Yes/No/Cancel also have extra methods allowing you to await the result. More information is below.

OK Dialog

```
await Dialogs.ShowMessage(App.Messenger, string message)
```

OK / Cancel Dialog

```
DialogOKCancelResponse response = await Dialogs.ShowOKCancel(App.Messenger,
                         string message,
                                       [OPTIONAL]string okText = "OK",
                                       [OPTIONAL]string cancelText = "Cancel");
```

Yes / No / Cancel Dialog

```
DialogYesNoCancelResponse response = await Dialogs.ShowYesNoCancel(App.Messenger,
                                              string message,
                                              [OPTIONAL]string yesText = "Yes",
                                              [OPTIONAL]string noText = "No",
                                              [OPTIONAL]string cancelText = "Cancel");
```

Progress Dialog

Indeterminate

The progress dialog can be used to run code in the background as you display an indeterminate progress dialog. To use the dialog while running code in the background, do the following -

```
await Dialogs.ShowProgress(App.Messenger,
         string message,
         Action codeToExecute)
```

Determinate

The alternative is to display the determinate progress bar and update the progress as you get it. The format in this case is a little different as you are not blocking on the line until the work is complete, you're actually updating the progress bar as it's open.

```
var dialog = new DialogProgressViewModel(string message,
                       [OPTIONAL]double min = 0,
                       [OPTIONAL]double max = 100);
dialog.Show(App.Messenger);

Task.Run(() =>
{
    for (int i = 0; i < 100; i++)
 {
        dialog.Value = i;
  DoWork();
 }

    dialog.CloseDialog();
});
```

Custom Dialogs

Custom Dialogs can be created by implementing the IDialog interface in your Dialog ViewModel.

Here is the class for IDialog for reference

```
public interface IDialog
{
    event Action<IDialog> DialogClosed;
    bool IsDialogOpen { get; set; }
    void OnDialogShow();
    Task WaitForDialogClose();
    void CloseDialog();
}
```

- `DialogClosed` is an event that is fired when the dialog is closed. This allows for asynchronous code to be run in the background while the dialog is open. If you attached to this event when displaying the dialog, you will be notified when it is closed.
- `IsDialogOpen` is used by the MainWindow to know when to display the dialog that has been passed.
- `OnDialogShow` is called when the Dialog is first opened.
- `WaitForDialogClose` is an async Task that can be used with await to stop the code on that line while displaying the Dialog without locking up the Application.
- `CloseDialog` can override waiting on the user to close the dialog by closing it manually.

There is a base class implementation of this interface to make it easier to build your own dialog ViewModel. It exists to shortcut some of the boilerplate code required in the dialog ViewModel. Here is the code of the base class, named `CustomDialogBase` -

```
public abstract class CustomDialogBase : PropertyChangedBase, IDialog
{
    public event Action<IDialog> DialogClosed;

    public bool IsDialogOpen { get; set; }

    public virtual void OnDialogShow() { }

    public void CloseDialog()
    {
        DialogClosed?.Invoke(this);
    }

    public virtual Task WaitForDialogClose()
    {
        // Task.Run throws this code onto a separate thread
        return Task.Run(() =>
        {
            while (IsDialogOpen)
            {
                System.Threading.Thread.Sleep(10);
            }
        });
    }
}
```

To show the custom dialog, use the following command -

```
await Dialogs.ShowCustom(App.Messenger, IDialog customDialog);
```

Notifications

To show Notifications you first must follow the steps in the Quick Start and Main Window ViewModel sections to add the `Messenger` and `NotificationHandler`.

The notifications were designed as an alternative way to notify the user without blocking the screen. Notifications should be used to display a short message to the user that something has happened. The events are obviously up to the developer, but the idea was to use Dialogs for essential events while Notifications are kind of a "thought you might like to know".

Notifications can be positioned to the top or bottom of the MainWindow by using the Property on `<controls:SFWindow/>`

```
NotificationPosition="Bottom"
NotificationPosition="Top"
```

There are 2 types of notification.

1. Notification with a countdown. This countdown time can be specified when being created. Once this time has elapsed the notification will be cleared. If the user clicks the notification before the time elapses, it will be closed.
2. Endless notifications. These will stay on screen until the user clicks them.

Both kinds of notification have an event that will be fired when they are closed. This gives you the chance to either log them or display them somewhere else on screen for the user to have record of.

The notification contains the following

- A count for the number of messages currently stacked up. Only one message will be displayed at a time.
- The message text itself.
- The amount of time left for the notification to display (in milliseconds).

The Notifications can be called from any ViewModel. This is done by sending a message to the Main Window of your application (in this case `MainWindow.xaml`).

To display a notification, use the following code

```
Notifications.Show(App.Messenger, string message);
Notifications.ShowWithTimeout(App.Messenger, string message, int timeout / in milliseconds)
```

To attach to the event for notifications being closed, use the following event in `MainWindow.xaml`. This code will create an event in the codebehind.

```
NotificationRemoved="MainWindow_NotificationRemoved"
```

To create an MVVM command for the event, use an Interaction Trigger (this can be used for any event in WPF).

```
xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"

<i:Interaction.Triggers>
    <i:EventTrigger EventName="NotificationRemoved">
        <ei:CallMethodAction MethodName="NotificationRemoved"
                             TargetObject="{Binding}" />
    </i:EventTrigger>
</i:Interaction.Triggers>
```

###Flyouts

Flyouts are overlay sections that can display user customized content over your main application window. The easiest way to explain is to show an example.

Flyouts by default will sit on top of the main window, but they can be Pinned and become part of the MainWindow layout as shown above.

There are 4 different flyouts, Left, Top, Right and Bottom and all can be given different User Controls. To attach a View to a flyout, use the following snippet. `FlyoutRight` shows that content can be placed directly inside the Flyout as opposed to using a `View` or `ContentPresenter`. All of this code resides inside `MainWindow.xaml`

```
 <controls:SFWindow.FlyoutLeft>
    <view:SingleFlyout Width="200"
                       DataContext="{Binding Flyouts.LeftFlyout}" />
</controls:SFWindow.FlyoutLeft>

<controls:SFWindow.FlyoutTop>
    <view:SingleFlyout Height="100"
                       DataContext="{Binding Flyouts.TopFlyout}" />
</controls:SFWindow.FlyoutTop>

<controls:SFWindow.FlyoutRight>
    <Grid Width="300">
        <StackPanel Background="{DynamicResource SFDefaultBackgroundBrush}">
            <Label Margin="5"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"
                   Content="Dismissed Notifications" />
            <ListBox ItemsSource="{Binding NotificationLog}"
                     ScrollViewer.HorizontalScrollBarVisibility="Disabled">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <Label FontSize="14">
                            <TextBlock Text="{Binding}"
                                       TextWrapping="Wrap" />
                        </Label>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </StackPanel>
    </Grid>
</controls:SFWindow.FlyoutRight>

<controls:SFWindow.FlyoutBottom>
    <view:SingleFlyout Height="100"
                       DataContext="{Binding Flyouts.BottomFlyout}" />
</controls:SFWindow.FlyoutBottom>
```

To show a flyout, use the `DependencyProperty` for that particular flyout.

```
PinFlyoutBottom="{Binding PinBottomFlyout}"
ShowFlyoutBottom="{Binding ShowBottomFlyout}"
```

With the matching properties in the ViewModel

```
private bool _showBottomFlyout;
public bool ShowBottomFlyout
{
    get { return _showBottomFlyout; }
    set
    {
        _showBottomFlyout = value;
        OnPropertyChanged();
    }
}

private bool _pinBottomFlyout;
public bool PinBottomFlyout
{
    get { return _pinBottomFlyout; }
    set
    {
        _pinBottomFlyout = value;
        OnPropertyChanged();
    }
}
```

Controls

Buttons

Standard button

```
<Button />
```

Toggle button

```
<ToggleButton />
```

Dark Theme button

```
<Button Style="{StaticResource SFButtonDark}" />
```

Icon Button

To use the icon button you need to first add a reference to the library's attached behaviors at the top of the class, like this

```
xmlns:wpfab="clr-namespace:WPFStylesLibrary.AttachedBehaviors;assembly=WPFStylesLibrary"
```

Then there are 3 attached behaviors for the regular button that allow you to use an icon. There's Icon, IconSize and IconPadding, so you should be able to get the perfect alignment by tweaking these values

IconSize defaults to 25 and IconPadding defaults to `"15,0,10,0`.

```
<Button wpfab:ButtonExt.Icon="{StaticResource IconCheck}"
  wpfab:ButtonExt.IconSize="20"
  wpfab:ButtonExt.IconPadding="20,0,20,0"
  Style="{StaticResource SFButtonIcon />
```

Icon Button Dark

To use the icon button you need to first add a reference to the library's attached behaviors at the top of the class, like this

```
xmlns:wpfab="clr-namespace:WPFStylesLibrary.AttachedBehaviors;assembly=WPFStylesLibrary"
```

Then there are 3 attached behaviors for the regular button that allow you to use an icon. There's Icon, IconSize and IconPadding, so you should be able to get the perfect alignment by tweaking these values

IconSize defaults to 25 and IconPadding defaults to `"15,0,10,0`.

```
<Button wpfab:ButtonExt.Icon="{StaticResource IconCheck}"
  wpfab:ButtonExt.IconSize="20"
  wpfab:ButtonExt.IconPadding="20,0,20,0"
  Style="{StaticResource SFButtonIconDark />
```

If you find a need for more icons to be added, we can either add them in the library or you can add them in your own project. The icons used should be from here to keep consistency.

The icons can be found in this resource dictionary

```
<ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Icons.xaml" />
```

When adding them, they should be in this format

```
<Canvas>
    <Path Data="M4.94,11.12C5.23,11.12 5.5,11.16 5.76,11.23C5.77,9.09 7.5,7.35 9.65,7.35C11.27,7.35 12.67,8.35 13.24,9.77
    Fill="#FF000000" />
</Canvas>
```

CheckBox

```
<CheckBox />
```

ComboBox

```
<ComboBox />
```

Radio Buttons

```
<RadioButton />
```

Progress Bar

```
<ProgressBar IsIndeterminate="False"
    Value="75" />
```

```
<ProgressBar IsIndeterminate="True" />
```

Slider

The slider control colors can be edited. By default they will range from light gray to the Accent color defined in `Accent.xaml`. If you wish to override these values, just set the `Foreground` and `Background` on the slider. The background is the color at 0%, while the foreground color is at 100%.

```
<Slider />
```

Progress Ring

Progress Ring is a custom control for a variation on the progress bar. It's primarily used for the Notification countdown, but can also be used as a separate control.

Along with the usual properties, here are some extra properties that should be set

- Radius - Needs to be set for the radius of the progress ring
- Time - The amount of time for the progress ring to be active. In milliseconds.
- IsIndeterminate
  - True - the progress ring continues indefinitely. The time property is not used in this scenario.
    - False - The progress ring runs once for **Time** milliseconds. To reset/restart the progress ring, IsActive must be toggled Off/On.
- IsActive - Starts the progress ring. When setting to False, the progress ring visibility is set to hidden.
- Completed [**EVENT**] - This event is called only when `Indeterminate="False"` and the time elapses.

```
<controls:SFProgressRing Radius="40"
       IsIndeterminate="False"
       Time="1000"
       IsActive="True"
       Foreground="Blue"
       Completed="ProgressRing_Completed" />
```

```
<controls:SFProgressRing Radius="40"
       IsIndeterminate="True"
       IsActive="True"
       Foreground="Green"
       Background="LightGray" />
```

NOTE: For the progress ring to smoothly fade to the background color, both `Foreground` and `Background` must be set appropriately. It will blend between these two colors.

ListView

```
<ListView />
```

TreeView

```
<TreeView />
```

TextBox

The `TextBox` has a couple of Attached Behaviors. To use these you have to add a reference to the `AttachedBehavior` namepace.

```
xmlns:ab="clr-namespace:WPFStylesLibrary.AttachedBehaviors;assembly=WPFStylesLibrary"

<TextBox ab:TextFieldExt.ClearTextButton="True"
         ab:TextFieldExt.Hint="Enter text only" />
```

The 2 extra attached behaviors `ab:TextFieldExt.ClearTextButton` and `ab:TextFieldExt.Hint` are self explanatory.

Data validation (see MVVM Library) code can be run on TextBox. To run the validation code you must add the Property (in this example, UserName) into the DataValidation collection with the following code.

```
public MainWindowViewModel()
{
    AddDataValidationForProperty(nameof(UserName), ValidateUserName);
}

private string ValidateContent()
{
    if (Regex.IsMatch(UserName, @"[^a-zA-Z\ ]"))
        return "Error, no numbers or special characters allowed!";

    else
        return null;
}
```

Then in the binding in the xaml code, you have to set `ValidatesOnDataError=True` and `NotifyOnValidationError=True`

```
<TextBox Text="{Binding UserName,
                        ValidatesOnDataErrors=True,
                        Mode=TwoWay,
                        UpdateSourceTrigger=PropertyChanged,
                        NotifyOnValidationError=True}"
         ab:TextFieldExt.ClearTextButton="True"
         ab:TextFieldExt.Hint="Enter text only" />
```

Here is what the validation looks like when the data is invalid.

PasswordBox

See TextBox, they are basically the same. The only additional properties allow you to bind the Password value to a SecureString. The standard PasswordBox doesn't have binding for security reasons, so this is a valid workaround.

`BindPassword` must be set to true for the `BoundPassword` property to work correctly.

```
<PasswordBox ab:PasswordBoxExt.BindPassword="True"
             ab:PasswordBoxExt.BoundPassword="{Binding Password,
                                                Mode=TwoWay,
                                                ValidatesOnDataErrors=True,
                                                NotifyOnValidationError=True,
                                                UpdateSourceTrigger=PropertyChanged}"
             ab:TextFieldExt.ClearTextButton="True"
             ab:TextFieldExt.Hint="Length &gt; 6" />
```

Number TextBox

See TextBox, again, they are very similar. This is a custom control that only allows for a type of number to be entered. This number can be a double, int, scientific notation or a custom format based on the `StringFormat`

- `HasDecimals` - shows or hides the decimal point and decimal digits

- `Delay` - changes the amount of time required before the control will start increasing or decreasing faster (when holding the buttons)
- `Value` - the number value of the control. This value is a **double**
- `ValueInt` - the number value of the control. This value is a **int** and the code is essentially (int)Value.
- `Interval` - the value change that occurs when the Plus or Minus buttons are pressed.
- `StringFormat` - sets the string formatting of the control.
  - N4 - shows 4 decimal places.

    - C2 - shows currency with 2 decimal places.
    - E1 - shows scientific notation with 1 decimal place.

    <controls:SFNumberTextBox Margin="5" VerticalAlignment="Center" Value="{Binding NumberValue}" Delay="500" Interval="{Binding ElementName=numberInterval, Path=Value}" HasDecimals="True" Speedup="True" StringFormat="{Binding ElementName=numberFormat, Path=Text}" />

Date Picker

The DatePicker should be bound to a `DateTime` object.

```
<DatePicker SelectedDate="{Binding Date}" />
```

Time Picker

The TimePicker should be bound to a `DateTime` object.

```
<controls:SFTimePicker SelectedTime="{Binding Time}" />
```

Color Picker/Canvas

The Color picker can bind to a Color property using `SelectedColor`. There are two Color Modes, `ColorCanvas` (on the left) and `ColorPalette` on the right.

```
<controls:SFColorPicker x:Name="colorpickerFill"
                        ColorMode="ColorCanvas"
                        SelectedColor="#FF0080FF" />
```

Overriding Accents

It is possible to easily restyle all the controls for a different accent color. This may be helpful for color coding the different areas of applications across Cardinal. To do this, you have to create your own Accent.xaml file and replace it in the correct areas.

This file can be found in the `WPFStylesLibrary` in the location `Styles\Accent.xaml`. Take all the code from the `Accent.xaml ResourceDictionary` and copy into a new `ResourceDictionary` called whatever you want. Then change the following snippet to use the new colors you would like to use.

```
<Color x:Key="SFAccentBrushColor100">#FF0080FF</Color>
<Color x:Key="SFAccentBrushColor80">#FF339AFF</Color>
<Color x:Key="SFAccentBrushColor60">#FF66B3FF</Color>
<Color x:Key="SFAccentBrushColor40">#FF99CCFF</Color>
<Color x:Key="SFAccentBrushColor20">#FFCCE6FF</Color>
```

Starting at `SFAccentBrushColor100` pick the color you would like to use with full brightness, then decrease the brightness value from 100% to 80%, 60%, 40% and 20% and enter them in to these resources.

ColorPicker.com is a great reference for doing this, you can select your color and then adjust the brightness value shown below.

In this example, the brightness value just needed to actually be set to 80, 60, 40 and 20 to get the appropriate values.

Once the file is finished, just replace the default line in `App.xaml` described in QuickStart

```
<ResourceDictionary Source="pack://application:,,,/WPFStylesLibrary;component/Styles/Accent.xaml" />
```

with your own

```
<ResourceDictionary Source="Styles/MySpecialAccent.xaml" />
```

Focus And Key Input

If you're planning on allowing the user to click buttons using they keyboard through shortcuts then there is an additional attached property that will come in use.

For the key input to be detected on your UserControl, it must have Focus. Unfortunately this doesn't happen automatically in WPF, so use the following code to automatically focus the user control whenever it's loaded -

First add a reference to the library's attached behaviors at the top of the class, like this

```
xmlns:wpfab="clr-namespace:WPFStylesLibrary.AttachedBehaviors;assembly=WPFStylesLibrary"
```

Now add this focus behavior to the top of the user control like the following example -

```
<UserControl x:Class="WPFStylesLibraryExample.View.Controls"
    xmlns:FocusExt.FocusOnLoad="True"
```

You shouldn't have to use this on Dialogs as they get passed focus when they are shown.

Also, this may already be known, but to bind the key input to a command, do the following in your user control -

```
<UserControl.InputBindings>
    <KeyBinding Command="{Binding DoSomethingImportantCommand}"
                Key="F1" />
</UserControl.InputBindings>
```