WPF with MVVM Light Toolkit

In the following guide I explain how you can build a simple Hello World WPF MVVM application. There are many MVVM toolkits out there, I prefer Laurent Bugnion's MVVM Light Toolkit for its light weight capabilities.

Creating a Hello World WPF MVVM Application

In Visual Studio, create a new WPF application project. Go to NuGet, search for and install "Mvvm Light" to the project. You'll see that a "ViewModel" folder was added with two classes inside it; `ViewModelLocator` and `MainViewModel`. Delete the latter class as it's just a sample, the former class however is the locator class that WPF uses to data bind views with their respective view models, so we'll need that. Create a "SenderViewModel" class and add it to the ViewModel folder:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GalaSoft.MvvmLight; //For mvvmlight
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Messaging;//for class Messenger
using wpfmvvm.Messages;


namespace wpfmvvm.ViewModel
{
  public class SenderViewModel: ViewModelBase
    {
        private String _textBoxText;
        public RelayCommand OnClickCommand { get; set; }

        public string TextBoxText
        {
            get
            { return _textBoxText; }

            set
            {
                _textBoxText = value;
                RaisePropertyChanged("TextBoxText");
            }
```

```
        }

        public SenderViewModel()
        {
            OnClickCommand = new RelayCommand(OnClickCommandAction,
null);
        }

        private void OnClickCommandAction()
        {
            var viewModelMessage = new ViewModelMessage()
            {
                Text = TextBoxText
            };
            Messenger.Default.Send(viewModelMessage);
        }

    }
}
```

Note the inheritance from `ViewModelBase`, and the public properties that we will be using to bind to our view. We are also using messaging here (which follows with the toolkit) to send messages from this sender view model. Create a folder called "Messages" and add a "ViewModelMessage" class to it:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GalaSoft.MvvmLight.Messaging;

namespace wpfmvvm.Messages
{
    class ViewModelMessage: MessageBase
    {
        public string Text { get; set; }
    }
}
```

Note the inheritance from `MessageBase`. Now go to `ViewModelLocator` and in the code rename "MainViewModel" to "SenderViewModel", build solution. Create a folder "View" and add a "SenderView" User Control (XAML) to it:

```
<UserControl x:Class="wpfmvvm.View.SenderView"
```

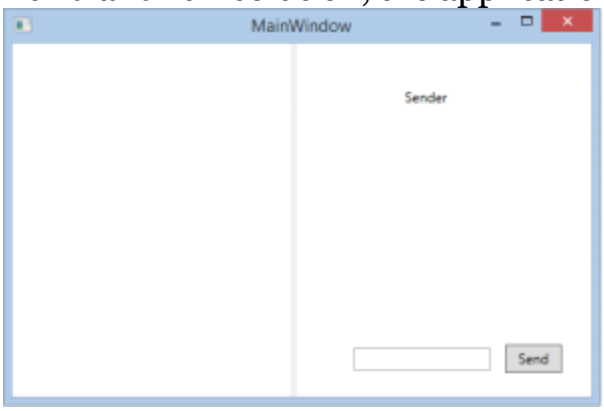
```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
             xmlns:Command="http://www.galasoft.ch/mvvmlight"
             xmlns:local="clr-namespace:wpfmvvm.View"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300"
DataContext="{Binding Source={StaticResource Locator},
Path=SenderViewModel}">
    <Grid>
        <Label Content="Sender" Margin="90,34,0,232"/>
        <TextBox HorizontalAlignment="Left" Width="120" Height="20"
Margin="50,266,0,11" Text="{Binding TextBoxText}"/>
        <Button Content="Send" Width="50" Height="25"
Margin="183,265,67,10">
            <i:Interaction.Triggers>
                <i:EventTrigger EventName="Click">
                    <Command:EventToCommand Command="{Binding
OnClickCommand}" />
                </i:EventTrigger>
            </i:Interaction.Triggers>
        </Button>
    </Grid>
</UserControl>
```

As you can see, `DataContext="{Binding Source={StaticResource Locator}, Path=SenderViewModel}` is where the magic happens. The data binding is done here, where the source refers to `ViewModelLocator` which has the key "Locator" that's automatically referenced inside `App`. Note how the text box element is bidden to the `TextBoxText` property that's defined in the view model. Finally, note how a trigger is defined to fire on the click event of the button element, bidden to the `OnClickCommand` relay command property. Now go to "MainWindow" and add the following XAML code inside the grid tags:

```
<Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
```

```
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <view:SenderView Grid.Row="0" Grid.Column="1" Loaded="SenderView_Loaded" />
        <GridSplitter HorizontalAlignment="Left" Width="5" Height="320"
Margin="245,0,0,-21"/>
    </Grid>
```

Build and run solution, the application will start:



We have finished the sender part, what remains now is the receiver. Create a "ReceiverViewModel" class and add it to the ViewModel folder:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GalaSoft.MvvmLight; //For mvvmlight
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Messaging;//for class Messenger
using wpfmvvm.Messages;

namespace wpfmvvm.ViewModel
{
    public class ReceiverViewModel : ViewModelBase
    {
        private string _contentText;

        public string ContentText
        {
            get { return _contentText; }
            set
            {
                _contentText = value;
                RaisePropertyChanged("ContentText");
            }
```

```
        }

        public ReceiverViewModel()
        {
            Messenger.Default.Register<ViewModelMessage>(this,
OnReceiveMessageAction);
        }

        private void OnReceiveMessageAction(ViewModelMessage obj)
        {
            ContentText = obj.Text;
        }
    }
}
```

Note how the receiving message is registered in the constructor. It is important to register for a message type prior to sending the message, otherwise the receiving view model won't receive the sent message, which makes sense. Now go to `ViewModelLocator` and register the view model you just created:

```
/*
  In App.xaml:
  <Application.Resources>
      <vm:ViewModelLocator xmlns:vm="clr-namespace:wpfmvvm"
                           x:Key="Locator" />
  </Application.Resources>

  In the View:
  DataContext="{Binding Source={StaticResource Locator},
Path=ViewModelName}"

  You can also use Blend to do all this with the tool's support.
  See http://www.galasoft.ch/mvvm
*/

using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Ioc;
using Microsoft.Practices.ServiceLocation;

namespace wpfmvvm.ViewModel
{
    /// <summary>
    /// This class contains static references to all the view models
in the
```

```
    /// application and provides an entry point for the bindings.
    /// </summary>
    public class ViewModelLocator
    {
        /// <summary>
        /// Initializes a new instance of the ViewModelLocator class.
        /// </summary>
        public ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() =>
SimpleIoc.Default);


            SimpleIoc.Default.Register<SenderViewModel>();
            SimpleIoc.Default.Register<ReceiverViewModel>();
        }

        public SenderViewModel SenderViewModel
        {
            get
            {
                return
ServiceLocator.Current.GetInstance<SenderViewModel>();
            }
        }
        public ReceiverViewModel ReceiverViewModel
        {
            get
            {
                return
ServiceLocator.Current.GetInstance<ReceiverViewModel>();
            }
        }


        public static void Cleanup()
        {
            // TODO Clear the ViewModels
        }
    }
}
```

Create a “ReceiverView” User Control and add it to the “View” folder:

```
<UserControl x:Class="wpfmvvm.View.ReceiverView"
```

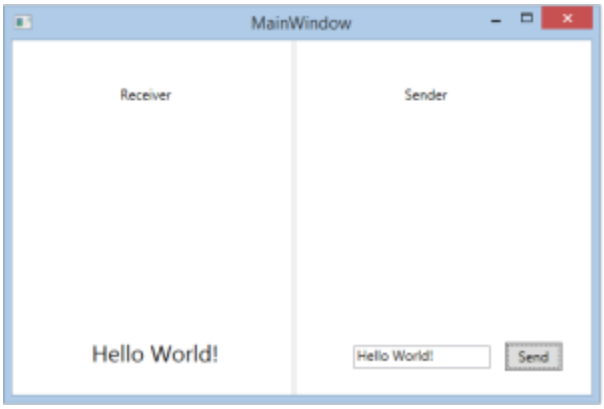
```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:wpfmvvm.View"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300"
DataContext="{Binding Source={StaticResource Locator},
Path=ReceiverViewModel}">
    <Grid>
        <Label Content="Receiver" Margin="90,34,0,232"/>
        <Label Content="{Binding ContentText}" FontSize="20"
Margin="65,255,40,0"/>
    </Grid>
</UserControl>
```

For the final step, go to `MainWindow` and include the `ReceiverView`:

```
<Window x:Class="wpfmvvm.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:wpfmvvm"
        xmlns:view="clr-namespace:wpfmvvm.View"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <view:ReceiverView Grid.Row="0" Grid.Column="0" />
            <view:SenderView Grid.Row="0" Grid.Column="1"
Loaded="SenderView_Loaded" />
            <GridSplitter HorizontalAlignment="Left" Width="5"
Height="320" Margin="245,0,0,-21"/>
```

```
        </Grid>
    </Grid>
</Window>
```

Build and run solution, the final product:



That's pretty much it. In this simple guide, we have demonstrated the use of the MVVM design pattern in WPF, something which any WPF developer should be comfortable to work with.

How to send string and content messages with MVVM Light Messenger

The **Messenger** component of MVVM Light easily allows to pass data between classes of our app:

?

```
1  // Sends a message with a Person object.
2  var person = new Person { FirstName = "Marco", LastName
   = "Minerva" };
3  Messenger.Default.Send(person);
4
5
6  // Registers for incoming Person messages.
7  Messenger.Default.Register<Person>(this, (person) =>
8  {
9      // Works with the Person object.
   });
```

To send an object, we simply need to pass it to the Messenger.Send method. But what if we just want to send string messages,
like GotoDetailsPage or CloseApp?

In this case, we don’t want to create a new class only to pass a string. We can instead use the NotificationMessage object that comes with MVVM Light:

```
1  // Sends a notification message with a string
2  content.
3  Messenger.Default.Send(new
   NotificationMessage("GotoDetailsPage"));
4
5  // Registers for incoming Notification messages.
6  Messenger.Default.Register<NotificationMessage>(this,
7  (message) =>
8  {
9      // Checks the actual content of the message.
10     switch (message.Notification)
11     {
12         case "GotoDetailsPage":
13             break;
14
15         case "OtherMessage":
16             break;
17
18         default:
19             break;
       }
   });
```

At lines 8-18, we check the Notification property to determine the specific message that has been sent. Of course, we can use constants to avoid the repetition of string values.
We have also a generic version of NotificationMessage that we can use when we want to send an object along with a string message, which for example specifies the action that we want to perform:

```
// Sends a notification message with a Person as content.
var person = new Person { FirstName = "Marco", LastName = "Minerva" };
Messenger.Default.Send(new NotificationMessage<Person>(person, "Select"));


// Registers for incoming Notification messages.
Messenger.Default.Register<NotificationMessage<Person>>(this, (message) =>
{
    // Gets the Person object.
    var person = message.Content;

    // Checks the associated action.
    switch (message.Notification)
    {
        case "Select":
            break;

        case "Delete":
            break;

        default:
            break;
    }
});
```

```
2
2

2
3
```

The generic version of NotificationMessage exposes a Content property that contains the actual object. So we can retrieve it and use the Notification value to determine which action to execute.