MVVM, WPF, and C#

Let’s start with our hands on example.I will create a simple WPF application

1) Launch Visual Studio 2017 (any edition will do). Choose C# as the development language and WPF Application from the available templates.

2) Choose a suitable name for your application. I have named it “WpfMVVM”

3) Add 3 folders in your solution with the names **View**,**Model**,**ViewModel**.

4) I will use a custom object to store the data so I will add a new class to the Model folder. I will name it **Footballer.cs** (yes, we will save and display footballer names). I will add some properties on this class like firstname,lastname,age,height,weight and another little property called SavedTime so I know when the data was last updated. So the first version of the **Footballer** class follows.

```
class Footballer
 {

     private string _firstname;

     public string FirstName
     {
         get { return _firstname; }
         set
         {
             _firstname = value;

         }
     }

     private string _lastname;

     public string LastName
     {
         get { return _lastname; }
         set
         {

             _lastname = value;

         }
     }

     private int _age;

     public int Age
     {
         get { return _age; }
         set
```

```
            {
                _age = value;

            }
        }

        private double _height;

        public double Height
        {
            get { return _height; }
            set
            {
                _height = value;

            }
        }

        private double _weight;

        public double Weight
        {
            get { return _weight; }
            set
            {
                _weight = value;

            }
        }

        private DateTime  _SavedTime;

        public DateTime  SavedTime
        {
            get { return _SavedTime; }
            set
            {
                _SavedTime = value;

            }
        }

}
```

Then I need to notify my UI (View) that some of these properties changed. I do that by implementing the **INotifyPropertyChanged** interface.

Now the **View** is notified when there changes in the property values,since when that happens notifications are fired. This allows the **View** to display any changes that have been made in the object's underlying properties. The second version of the **Footballer** class follows

```
class Footballer:INotifyPropertyChanged
{

private string _firstname;

public string FirstName
{
get { return _firstname; }
set
{
_firstname = value;
OnPropertyChanged("FirstName");

}
}

private string _lastname;

public string LastName
{
get { return _lastname; }
set
{

_lastname = value;
OnPropertyChanged("LastName");
}
}

private int _age;

public int Age
{
get { return _age; }
set
{
_age = value;
OnPropertyChanged("Age");
}
}
private double _height;

public double Height
{
get { return _height; }
set
{
_height = value;
OnPropertyChanged("Height");
}
}

private double _weight;

public double Weight
{
get { return _weight; }
set
{
_weight = value;
OnPropertyChanged("Weight");
}
}

private DateTime _SavedTime;

public DateTime SavedTime
{
get { return _SavedTime; }
set
{
_SavedTime = value;
OnPropertyChanged("SavedTime");
}
}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(string property)
{
if (PropertyChanged !=null)
{
PropertyChanged(this, new PropertyChangedEventArgs(property));
}

}

}
```

If I want to implement data validation logic, the **Model** class is a good place to do it.In this case I need to implement the **IDataErrorInfo** interface.In this example I will add some validation for the fields “FirstName”, “Lastname”. These fields cannot be empty.

The third version of the **Footballer** class follows

```
class Footballer:INotifyPropertyChanged,IDataErrorInfo
{

private string _firstname;

public string FirstName
{
get { return _firstname; }
set
{
_firstname = value;
OnPropertyChanged("FirstName");

}
}

private string _lastname;

public string LastName
{
get { return _lastname; }
set
{

_lastname = value;
OnPropertyChanged("LastName");
}
}

private int _age;

public int Age
{
get { return _age; }
set
{
_age = value;
OnPropertyChanged("Age");
}
```

```
}

private double _height;

public double Height
{
get { return _height; }
set
{
_height = value;
OnPropertyChanged("Height");
}
}

private double _weight;

public double Weight
{
get { return _weight; }
set
{
_weight = value;
OnPropertyChanged("Weight");
}
}

private DateTime _SavedTime;

public DateTime SavedTime
{
get { return _SavedTime; }
set
{
_SavedTime = value;
OnPropertyChanged("SavedTime");
}
}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(string property)
{
if (PropertyChanged !=null)
{
PropertyChanged(this, new PropertyChangedEventArgs(property));
}

}

public string Error
{
```

```
get { return null; }
}

public string this[string columnName]
{
get

{
string theerror = string.Empty;

if ((string.IsNullOrEmpty (_firstname)))
{
theerror = "This field is required";
}

else if ((string.IsNullOrEmpty(_lastname)))
{
theerror = "This field is required";
}
return theerror;
}
}

}
```

We will have to make some changes to the XAML (Set the ValidatesOnDataErrors=True for firstname and lastname elements) for this to work but I will show you later.

Now let’s move to the **ViewModel** implementation. This model must have properties that expose instances of the **Model**objects.
So in this case we must have a property in the **ViewModel** class that exposes the Footballer class that lives in the **Model**.
Add a new item to your project, a class file, name it **FootballerViewModel.cs**

The first thing I need to do in my new class is to implement the **INotifyPropertyChanged** interface.This is because the **View** is bound to the **ViewModel**. **INotifyPropertyChanged** interface is the way to push data to the **View**. The first version of the **ViewModel** class follows

```
class FootballerViewModel:INotifyPropertyChanged
{

public event PropertyChangedEventHandler  PropertyChanged;

public void OnPropertyChanged(string propertyName)
{
if (PropertyChanged != null)
{
PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

}
}
```

Let’s create a property that is an instance of the **Model** object .The second version of the **ViewModel**class follows

```
class FootballerViewModel:INotifyPropertyChanged
{

private Footballer  _myfootballer;

public Footballer MyFootballer
{
get {  return _myfootballer; }
set

{

_myfootballer = value;
OnPropertyChanged("MyFootballer");

}
}

public event PropertyChangedEventHandler  PropertyChanged;

public void OnPropertyChanged(string propertyName)
{
if (PropertyChanged != null)
{
PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

}
}

```

Now I will use the constructor for the **ViewModel** class and populate it with some random data.

The third version of the **ViewModel** follows

```
class FootballerViewModel:INotifyPropertyChanged
{

public FootballerViewModel()
{
LoadFootballers();
}

private void LoadFootballers()
{
MyFootballer  = new Footballer()
{
FirstName = "Steven",
LastName = "Gerrard",
Age = 31,
Weight = 88.6,
Height = 1.84

};
}

private Footballer _myfootballer;

public Footballer MyFootballer
{
get { return _myfootballer; }
set

{

_myfootballer = value;
OnPropertyChanged("MyFootballer");

}
}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(string propertyName)
{
if (PropertyChanged != null)
{
```

```
PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

}
}
```

Now, let’s create the **View**. The **View** will be purely **XAML.** It consists of the visual elements like buttons,textboxes. It is not responsible for retrieving data,implement business logic or validation logic.

First things first. Move the **MainWindow.xaml** file to the **View** folder.

Go to the **App.xaml** file and change the **StartupUri** attribute of the **Application** element to ***StartupUri=”View/MainWindow.xaml”***

```
<Window x:Class="WpfMVVM.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525" >

<Grid>

<Grid.RowDefinitions>
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
<RowDefinition Height="46*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="121*" />
<ColumnDefinition Width="382*" />
</Grid.ColumnDefinitions>
<TextBlock Height="23" HorizontalAlignment="Left" Margin="10,12,0,0" Name="textBlock1" T
Width="82" FontSize="16" />
<TextBox Grid.Column="1" Height="23" HorizontalAlignment="Left" Margin="45,12,0,0" Name=
Width="266" />
<TextBlock Grid.Row="1" Height="23" HorizontalAlignment="Left" Margin="12,8,0,0" Name="t
VerticalAlignment="Top" Width="82" FontSize="16" />
<TextBlock Grid.Row="2" Height="23" HorizontalAlignment="Left" Margin="12,9,0,0" Name="t
VerticalAlignment="Top" FontSize="16" />
<TextBlock Grid.Row="3" Height="23" HorizontalAlignment="Left" Margin="10,12,0,0" Name="
VerticalAlignment="Top" FontSize="16" />
<TextBlock Grid.Row="4" Height="23" HorizontalAlignment="Left" Margin="10,9,0,0" Name="t
```

```
VerticalAlignment="Top" FontSize="16" />
<TextBox Grid.Column="1" Grid.Row="1" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266"  />
<TextBox Grid.Column="1" Grid.Row="2" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266"  />
<TextBox Grid.Column="1" Grid.Row="3" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266"  />
<TextBox Grid.Column="1" Grid.Row="4" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266"  />
<TextBlock Grid.Row="5" HorizontalAlignment="Left" Margin="12,12,0,13" Name="textBlock6
<TextBox Grid.Column="1" Grid.Row="5" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" />
<Button Content="Update" Grid.Column="1" Grid.Row="6" Height="24" HorizontalAlignment="
Margin="92,15,0,0" Name="button1" VerticalAlignment="Top" Width="156" FontSize="16" Font
FontWeight="Bold" BorderBrush="#FF407F2E" Foreground="#FFE39223" Background="#FF175D17'
</Grid>
</Window>
```

Now let’s make the necessary changes on the **View** so it can bind to the **Model** through the **ViewModel**.

We have first to make a reference to theViewModel class/object so I include the following namespace.

```
xmlns:football="clr-namespace:WpfMVVM.ViewModel"
```

This is the first step to hook a **View** to the **ViewModel.** We tell WPF where our **ViewModel** lives**.**

Then I will create the **ViewModel** as a resource (static resource).

```
<Window.Resources>

        <football:FootballerViewModel x:Key="Footb
allerViewModel" />

</Window.Resources>
```

Then in the **Grid** element I make the following change

```
Grid DataContext="{StaticResource
FootballerViewModel}" >
```

The **DataContext** property **inherits its value to child elements**. So you can set the ***DataContext*** on a layout container (Grid) and its values are inherited to all child elements. This is very useful in our case where we want to bind to the same data object.

Databinding is **typically done in XAML** by using the **{Binding}** markup extension. We have to bind the **Text** attribute of the **TextBox** element to the various properties as implemented in the **Model** through the **ViewModel**.

```
<TextBox Grid.Column="1" Height="23" HorizontalAlignment="Left" Margin="45,12,0,0" Name=
VerticalAlignment="Top" Width="266" Text="{Binding MyFootballer.FirstName ValidatesOnD

<TextBox Grid.Column="1" Grid.Row="1" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" Text="{Binding MyFootballer.LastName, ValidatesOnD

<TextBox Grid.Column="1" Grid.Row="2" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" Text="{Binding MyFootballer.Age}" />

<TextBox Grid.Column="1" Grid.Row="3" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" Text="{<strong>Binding MyFootballer.Height</strong

<TextBox Grid.Column="1" Grid.Row="4" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" Text="{Binding MyFootballer.Weight}" />

<TextBox Grid.Column="1" Grid.Row="5" Height="23" HorizontalAlignment="Left" Margin="45,
VerticalAlignment="Top" Width="266" Text="{Binding MyFootballer.SavedTime}" />
```

Run your application and see the values from the domain object appearing in the textboxes.

We do have a button in our UI. We need to update data from the UI and bind the new data to the textboxes.

We need to have a different communnication between the **View** and the **ViewModel**. In this case we will use **Commands**.

First we need to update the XAML.

```
<Button Content="Update" Grid.Column="1" Grid.Row="6" Height="24" HorizontalAlignment="Le
Name="button1" VerticalAlignment="Top" Width="156" FontSize="16" FontFamily="Garamond"
FontWeight="Bold" BorderBrush="#FF407F2E" Foreground="#FFE39223" Background="#FF175D17"
Command="{Binding SaveFootballerCommand}" />
```

So I updated the XAML for the button element by adding the *Command="{Binding SaveFootballerCommand}"*

Now we must add some more code in our **ViewModel**.

```
public class SaveFootballerCommand:ICommand
{

Action _saveMethod;

public bool CanExecute(object parameter)
{
return true;
}

public event EventHandler CanExecuteChanged;

public void Execute(object parameter)
{
_saveMethod.Invoke();
}
}
```

I create a new class, SaveFootballerCommand, that has to implement the **ICommand** interface. Then I define an Action, *_saveMethod*, which is what is executed when the button is clicked.

The *CanExecute* tells me if I am allowed to execute the command.Then I create the *Execute* method and invoke it.

Now we need to create a property in my **ViewModel** that exposes that command.

```
private ICommand _SaveFootballerCommand;

public ICommand SaveFootballerCommand
{
get { return _SaveFootballerCommand; }
set
{
_SaveFootballerCommand = value;
OnPropertyChanged("SaveFootballerCommand");

}
}
```

Now we need to create an instance of the *SaveFootballerCommand*.

```
private void FireCommand()
{
SaveFootballerCommand = new SaveFootballerCommand(updateFootballer) ;
}
```

I place the inside my constructor the **FireCommand()** method.

```
public FootballerViewModel()
{
FireCommand();
LoadFootballers();
}
```

The *updateFootballer* method is going to be invoked when the command is executed.So I must implement it.

```
private void updateFootballer()
{

MyFootballer.SavedTime = DateTime.Now;

}
```

Now I need to create a constructor of the **SavePersonCommand** class that takes a parameter. The code follows

```
public SaveFootballerCommand(Action updateFootballer)
{
_saveMethod = updateFootballer;

}
```

The whole code for the ViewModel class follows

```
public SaveFootballerCommand(Action updateFootballer)
{
class FootballerViewModel:INotifyPropertyChanged
{

public FootballerViewModel()
{
FireCommand();
LoadFootballers();
}

private void FireCommand()
{
SaveFootballerCommand = new SaveFootballerCommand(updateFootballer) ;
}

private void updateFootballer()
{

MyFootballer.SavedTime = DateTime.Now;

}

private void LoadFootballers()
{
MyFootballer  = new Footballer()
{
FirstName = "Steven",
LastName = "Gerrard",
Age = 31,
Weight = 88.6,
Height = 1.84

};
}
private Footballer _myfootballer;

public Footballer MyFootballer
{
get { return _myfootballer; }
set

{

_myfootballer = value;
OnPropertyChanged("MyFootballer");

}
}

private ICommand _SaveFootballerCommand;

public ICommand SaveFootballerCommand
{
get { return _SaveFootballerCommand; }
set
{
_SaveFootballerCommand = value;
OnPropertyChanged("SaveFootballerCommand");

}
}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(string propertyName)
{
if (PropertyChanged != null)
{
PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

}
}

}
```

Run your application and click the button.Note how the date is updated.

Note that we do not have written a single line of code to the **MainWindow.xaml.cs** file.
Now that we have a clear understanding of what **MVVM** is and what the different components are, let's review again the main concepts.
The **ViewModel** is responsible for the state and behaviour of the **View** and it acts like the "middle man" between the **View** and the **Model**.
It the middle layer and its main job is to send **View** information to
the **Model** and **Model** information to the **View**.
It has no dependency on the **View** so we can reuse the **ViewModel** on different **Views** or even other platforms.
The **ViewModel** exposes the **Model** as **properties** or **commands**.
In any case it must implement the **INotifyPropertyChanged** interface.
The **View** binds to properties in the **ViewModel**. It does that by setting the **DataContext** of a **View** to an instance of the **ViewModel** object.
The **Model** is tha data.Simple as that. The Model's job is to represent the data and has no knowledge of where or how the data will be presented.
By using this pattern we can have the designers in our company designing the Views with xaml using Visual Studio Designer or Blend. The developers in our company can work with the **Model** and the **ViewModel** writing the data access code and businnes logic . We reduce the development time.
This is great pattern because we can make our code more **testable** and **maintainabl**