طريقك إلى MVVM

بسم الله الرحمن الرحيم

سنقوم هنا بعرض طريق مختصر وصغير لتقنية MVVM الرائعة، لذلك لن نطيل في الشرح النظري لكي لا نفقد متعة الجزء العملي والتطبيقي، بالتالي ستكون المقدمة مختصرة قدر الإمكان، وسنترك التوضيح للجزء العملي. وبعد أن نسلك هذا الطريق الصغير، ستكون لدينا القدرة للخوض في الطرقات الأخرى المعقدة والممتعة بنفس الوقت.

ما هي تقنية MVVM؟

تقنية من ميكروسوفت تستخدم لبناء التطبيقات ذات الواجهات (Presentation Application)، تعتمد وبشكل كبير على WPF, Silverlight، حيث أنها تستهلك معظم خواص تلك التقنيات مثل (Data Binding, Command, Styles, Templates…). اشتق الإسم من (Model-View-ViewModel)، بالتالي فهي مكونة من ثلاث أقسام رئيسية، سنوجزها بإختصار.

Model:

في عالم البرمجة مصطح مودل مقابل لمصطلح بيانات (Data)، بالتالي ضمن تطبيق MVVM فإن Model يمثل البيانات التي سنتعامل معها، طبعاً ليس كل أنواع البيانات. مثلاً لو كان لدينا تطبيق قواعد بيانات، فإن الكلاسات (Classes) التي تعكس حقول الجداول، هي ما يطلق عليها Models. والأهم من هذا أن تكون هذه المودلز مستقلة عن التطبيق ولا تأخذ أي اعتبار للأجزاء الأخرى، بمعنى لا يجب أن ترتبط أو تؤشر على جزء أخر من التطبيق، على العكس الأجزاء الأخرى ترتبط معها وتؤشر عليها.

View:

أيضاً مصطلح View مقابل لمصطلح الواجهات User Interfaces، لكن ضمن MVVM ليست هي تماما الواجهات إذا يمكن أن تمثل View وحدة، لقطعة صغيرة من واجهة معينة، بالتالي ربما تحوي الواجهة الواحدة على View أو أكثر. الجوهو الأساسي في بناءها هو الـ Data Binding، فضمن MVVM لا يوجد كود للواجهة(Code Behind). بالتالي ايضاً لا فهي مستقلة عن أجزاء التطبيق الأخرى.

ViewModel:

ألية الربط بين الـ Model والـ View، وهي ما تجعل كل منها مستقل عن التطبيق، حيث تقوم الـ ViewModelبتنظيم خواص (Properties) الخاصة بمودل معين، وتظهرها ل View بشكل تستطيع عرض تلك البيانات بشكل User Controls. كما توفر ألية لتنفيذ العمليات والأحداث على تلك البيانات، أقرب مثال هو تنفيذ حدث زر معين. حيث تقوم الـ ViewModel بعرض خاصية من نوع ICommand مسؤولة عن تنفيذ حدث معين، بينما يقصر عمل الـ View على ربط ذلك الزر بهذه الخاصية، عن طريق DataBinding كما تحدثنا.

متطلبات العمل مع MVVM؟

بالطبع لا يمكن البدء بتعلم هذه التقنية الرائعة دون المرور والتعرف على مجموعة من الميزات الرائعة أيضاً ضمن بيئة العمل، بما أن بداية هذا الطريق ستكون سهلة، لذلك التعرف على هذه الميزات سيكون أسهل، حيث سنقدم مبادئ بسيطة عن كل ميزة، خلال التطبيق، وفي مراحل التطوير سنتعرف على الخواص الأخرى والطرق الأخرى في استخدام تلك الميزات، والتي أهمها:

1. Data Binding
2. Commands
3. Styles
4. Templates
5. Validation
6. Notifications
7. Converters

الآن دعونا نبدأ بالجزء العملي، وسنلاحظ خلال التطبيق فائدة تلك الميزات، حيث سنقوم بعرضها بأبسط طريقة كما تحدثنا، وعند الوصول لنقطة معينة، سنقوم بعرض أفضل طريقة لاستخدام ميزة معينة.

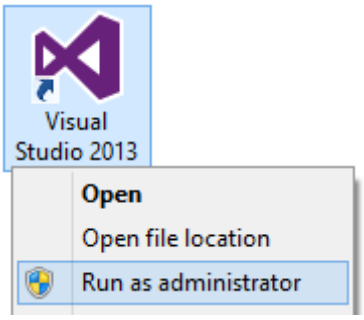
للحصول على فهم أفضل، قم بمتابعة وتطبيق الخطوات التالية خطوة بخطوة، بالطبع يوجد تطبيق هذا الدرس في المرفقات.

بناء نظام بيانات مصغر

الهدف من هذا المشروع الصغير، هو بناء تطبيق صغير يقوم بعرض مجموعة من الأشخاص People حيث كل شخص يمثل بالمودل Person، وبإمكاننا إضافة شخص جديد، تعديل أو حذف شخص سابق، بالطبع لن نعتمد على قواعد البيانات، لكي لا نضيف درجة تعقيد إضافية للمشروع (حيث أن استخدام قواعد البايانات هي هدفنا من هذه الدروس لاحقاً بإذن الله تعالى).

سيتم تنفيذ المشروع اعتماد على مفهوم التطوير السريع (Agile Methodology)، اي سنقوم ببناء تطبيق صغير، ثم سنقوم بتوسعته شيئاً فشيئاً، ليتسنى لنا فهم كل جزء بشكل أفضل، لذلك في النهاية سيكون لدينا عدة إصدارات من هذا العمل.

1. قم بتشغيل الـ Visual Studio بشكل مسؤول، وذلك بالضغط على أيقونته بالزر الأيمن واختيار خيار(Run As Administrator).



ثم قم بإنشاء مشروع جديد من نوع (WPF Application)، أعطيه إسم (Mvvm.Session01)، ولا تنسى أيضاً تسمية المشروع (Solution) بإسم مناسب مثل (MvvmWay).

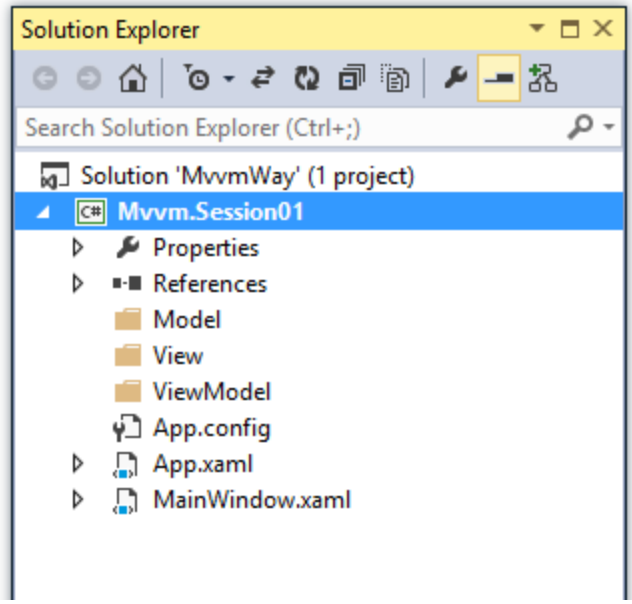
2. قم بإضافة ثلاث مجلدات إلى مشروعك الجديد بالإسماء التالية.

(1 Model

(2 View

(3 ViewModel

الآن يجب أن يكون لديك شكل المشروع بالشكل التالي:



3. الأن قم بإضافة كلاس جديد (New Class) إلى مجلد Model، سميه (Person.cs)، وقم بكتابته بالشكل التالي:

```
namespace Mvvm.Session01.Model
{
    public class Person
    {
        private int _id;
        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        private string firstName;
        public string FirstName
        {
            get { return firstName; }
            set { firstName = value; }
        }

        private string lastName;
        public string LastName
        {
            get { return lastName; }
            set { lastName = value; }
        }

        public string FullName
        {
            get
            {
```

```
                return string.Format("{0} {1}", FirstName, LastName);
            }
        }
    }
}
```

4. قم بإضافة كلاس جديد تحت إسم (PersonViewModel.cs) ضمن مجلد ViewModel، قم بكتابته بالشكل التالي:

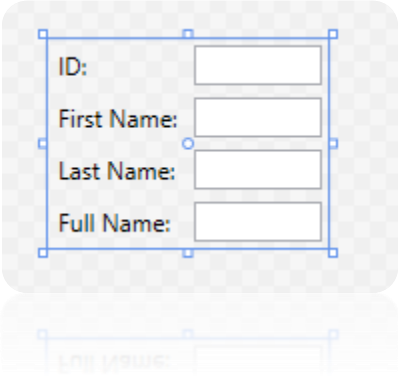
```
namespace Mvvm.Session01.ViewModel
{
    using Mvvm.Session01.Model;
    public class PersonViewModel
    {
        private Person model;

        public int Id
        {
            get { return model.Id; }
            set
            {
                model.Id = value;
            }
        }

        public string FirstName
        {
            get { return model.FirstName; }
            set
            {
                model.FirstName = value;
            }
        }

        public string LastName
        {
            get { return model.LastName; }
            set
            {
                model.LastName = value;
            }
        }
        public string FullName
        {
            get { return model.FullName; }
        }
        public PersonViewModel(Person person)
        {
            model = person;
        }

        public PersonViewModel()
            :this(new Person())
        { }
```

```
    }
}
```

5. الأن قم بإضافة عنصر من نوع User Control إلى مجلد View، وأعطه إسم Person.xaml، كما تحدثنا سنستخدم هذا الجزء لعرض البيانات للمستخدم، بالتالي يجب علينا الإهتمام بتنظيم وتركيب هذا الجزء، ففي النهاية المستخدم لن يرى سواه.

الآن قم بتحديث كود XAML ليصبح كالتالي:

```xaml
<UserControl x:Class="Mvvm.Session01.View.Person"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <UniformGrid Columns="2">
        <Label Content="ID:"/>
        <TextBox Text="{Binding Id}"/>

        <Label Content="First Name:"/>
        <TextBox Text="{Binding FirstName}"/>

        <Label Content="Last Name:"/>
        <TextBox Text="{Binding LastName}"/>

        <Label Content="Full Name:"/>
        <TextBox Text="{Binding FullName, Mode=OneWay}"/>
    </UniformGrid>
</UserControl>
```



الآن قد يخطر لك تساؤل كيف سيتم ربط الأجزاء الثلاثة ببعضها البعض؟

كما لاحظنا فإن الـ ViewModel قامت بتعريف مؤشر على اوبجكت من نوع Person وهو ما ستستخدمه كـ Model لتنظيم البيانات، وأيضاً قامت بعمل تغليف لخواص ذلك المودل، حيث قمنا بتعريف Id, FirstName, LastName, FullName مرة أخرى عن طريق model. بالتالي يبقى لدينا أن نربط الـ

View بـِViewModel ، وعندها تتمكن الـ View من الدخول للخواص السابقة مباشرة (طبعاً هذا المفهوم هو أحد المفاهيم التي سنتوسع بشرحها لاحقاً، فهل من المعقول إعادة كتابة جميع الـ Properties ضمن الـ ViewModel؟ وسنعطي الحلول المتوفرة ووجهات النظر حولها، لكن الآن يكفينا النظرة الحالية).

أما عن كيفية ربط الـ View بـِ ViewModel فيتم عن طريق الخاصية DataContext الموجود ضمن FrameworkElement، وبما أن الـ View من نوع User Control والذي بدوره حفيد ل FrameworkElement ، بالتالي إسناد هذه الخاصة لأوبجكت جديد من نوع PersonViewModel، كفيل بتحقيق الهدف.

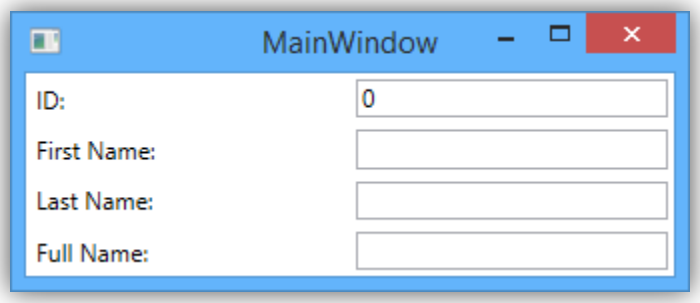
6. الآن ضمن الكود الخاص بـ View (Code Behind) نقوم بكتابة السطر التالي، بالطبع هذه الطريقة خاطئة نوعاً ما، فهناك عدة طريق لتحقيق هذا الهدف، ولكننا نكتفي بالأبسط والأوضح هنا:

```
namespace Mvvm.Session01.View
{
    using System.Windows.Controls;
    using Mvvm.Session01.ViewModel;
    public partial class Person : UserControl
    {
        public Person()
        {
            InitializeComponent();
            DataContext = new PersonViewModel();
        }
    }
}
```

7. الآن لجعل التطبيق قابل للعمل، يجب استضافة الـ View ضمن نافذة معينة، لذلك سنستخدم النافذة الأساسية MainWindow.xaml، حيث نقوم بتعريف فضاء أسماء جديد ضمنها للإشارة إلى مجلد الـ View، ضمن نضيف عنصر من نوع Person.xaml إلى الواجهة بالشكل التالي:

```
<Window x:Class="Mvvm.Session01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:Views="clr-namespace:Mvvm.Session01.View"
        Title="MainWindow" Height="140" Width="340">
    <Grid>
        <Views:Person/>
    </Grid>
</Window>
```

8. الآن نقوم بتشغيل البرنامج بالضغط على F5، لتظهر لدينا الواجهة كالتالي:



ربما تتساءل ما فائدة هكذا برنامج لا يسمن ولا يغني من جوع؟

في الواقع وكما تحدثت أن عملية التطوير ستكون بشكل إصدرات، وهذا الإصدار سيكون طريقنا للتعرف على الميزات الأخرى والتي ستضفي ميزات كبيرة على تطبيقنا، لذلك سنبدأ بمناقشة بعض المشاكل، والتي سينتج عن حلها التعرف على ميزات جديدة.

Notification System

كان من المفترض عند إدخال قيم ل FirstName, LastName أن يتم تحديث قيمة الـ FullName، وقد لاحظنا أن الخاصية FullName ضمن المودل Person عبارة عن الإسم الأول متبوع بمسافة بيضاء ثم الإسم الأخير، لكن شيء من هذا لم يحدث!

الطريقة التقليدية كانت بإعادة إسناد خاصية الإسم الكامل للحقل FullNameTextBox، وهذا صحيح، لكننا الآن ضمن نظام منفصل الطبقات ويجب فصل عملية العرض عن عملية الـ Loigc. لذلك يجب وجود آلية مهتمها إبلاغ FullName TextBox بضرورة تحديث قيمتها، بسبب حدوث تغير بإحدى مكوناتها (FirstNam or LastName). هذه الآلية هي Notification ويتم تطبيقها ضمن ViewModel أو Model باعتبارهم مصدر البيانات، عن طريق تطبيق INotifyPropertyChanged Interface، ومن خلاله يمكننا تنبيه نظام DataBinding بضرورة تحديث البيانات اللازمة.

تطبيق عملية التنبيه ضمن Model أو ViewModel تختلف حسب طبيعة المشروع وتركيبته، نحن الآن نملك الخيار، سنضعه ضمن ViewModel.

لذلك سنجعل الكلاس PersonViewModel يرث من INotifyPropertyChanged والموجود ضمن فضاء الأسماء System.ComponentModel، ستعطينا هذه العملية حدث وحيد يسمى PropertyChanged من نوع PropertyChangedEventHander، لذلك عند حدوث تعديل معين في البيانات، نقوم بتشغيل هذا الحدث.

عملية تشغيل الحدث تكون كالتالي:

```
if (PropertyChanged != null)
     PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
```

ومكانها سيكون في مرحلة إسناد البيانات ل FirstName, LastName أي ضمن الـ set كالتالي:

```
public string FirstName
{
     get { return model.FirstName; }
     set
     {
         model.FirstName = value;
         if (PropertyChanged != null)
             PropertyChanged(this, new PropertyChangedEventArgs("FullName"));
       }
}
```

لكن ولكي لا نعيد كتابة السطر الممل السابق، سنقوم بوضعه ضمن ميثود خاصة، نسميها RaisePropertyChanged، ونعطيها إسم الخاصية التي حدث عندها التغيير، فتكون لدينا بالشكل التالي:

```
private void RaisePropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
```

بعد هذه المجموعة من التعديلات يصبح لدينا شكل الـ PersonViewModel.cs بالكامل كالتالي:

```
namespace Mvvm.Session01.ViewModel
{
    using Mvvm.Session01.Model;
    using System.ComponentModel;
    public class PersonViewModel : INotifyPropertyChanged
    {
        private Person model;

        public int Id
        {
            get { return model.Id; }
            set
```

```
            {
                model.Id = value;
            }
        }

        public string FirstName
        {
            get { return model.FirstName; }
            set
            {
                model.FirstName = value;
                RaisePropertyChanged("FullName");
            }
        }

        public string LastName
        {
            get { return model.LastName; }
            set
            {
                model.LastName = value;
                RaisePropertyChanged("FullName");
            }
        }
        public string FullName
        {
            get { return model.FullName; }
        }
        public PersonViewModel(Person person)
        {
            model = person;
        }

        public PersonViewModel()
            :this(new Person())
        { }

        #region INotifyPropertyChanged Members

        public event PropertyChangedEventHandler PropertyChanged;

        private void RaisePropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
        #endregion
    }
}
```

الآن نقوم بتشغيل التطبيق F5، ونقوم بإدخال الإسم الأول والإسم الأخير، ولكن نلاحظ ملاحظة بسيطة:

الملاحظة أنه لا يتم تحديث الإسم الكامل إلا بعد فقد التركيز من الحقل، مثلاً كما نلاحظ في الصورة، تم تحديث الإسم الكامل إلى الإسم الأول فقط! لأننا مازلنا داخل حقل الإسم الأخير، وبمجرد تغيير التركيز من حقل الإسم الأخير سنحصل على الإسم الكامل كما يفترض.

هنا ينطرح سؤال؟ لماذا هذا السلوك، وهل يمكن تغييره؟

سبب هذا السلوك أنه في بعض الأحيان ربما يكون تحديث القيم، يعتمد على عملية طويلة نسبياً، كالبحث في قاعدة البيانات، فليس من الضروري تحديث قيمة معينة عند كتابة كل حرف، مما سيؤخر عمليه التنفيذ، بل عند كتابة القيمة كاملة، نقوم بالتحديث لمرة واحدة، وهذا سنلاحظة في بعض الأجزاء لاحقاً، أما الآن وبما أننا بحاجة لتحديث الإسم الكامل عند الكتابة مباشرة، نقوم بالتحديث على عملية `DataBinding` حيث نطلب منها القيام بالتحديث فور الكتابة، لذلك نذهب على `Person.xaml` ونقوم بتحديثها بالشكل التالي:

```
<UserControl x:Class="Mvvm.Session01.View.Person"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <UniformGrid Columns="2">
        <Label Content="ID:"/>
        <TextBox Text="{Binding Id}"/>

        <Label Content="First Name:"/>
        <TextBox Text="{Binding FirstName, UpdateSourceTrigger=PropertyChanged}"/>

        <Label Content="Last Name:"/>
        <TextBox Text="{Binding LastName, UpdateSourceTrigger=PropertyChanged}"/>

        <Label Content="Full Name:"/>
        <TextBox Text="{Binding FullName, Mode=OneWay}"/>
    </UniformGrid>
</UserControl>
```

الآن قم بتشغيل البرنامج ولاحظ السلوك المطلوب.

هذا بالنسبة لعنصر واحد (One Object)، ماذا لو أردنا عرض مجموعة من العناصر(List of Object)؟

أصبح واضحٌ لدينا أن جوهر العملية يتم عن طريق إنشاء ViewModel والذي ستعتمد عليه الـ View لعرض البيانات والتفاعل مع المستخدم، إذن ما نحن بحاجته هو مجموعة من التحسينات على الـ ViewModel بما يناسب المتطلب.

لذلك لو أردنا عرض مجموعة من Person نحن بحاجة لتعريف خاصية (Property) ضمن الـ ViewModel من نوع IList, IEnumerable، ونملؤها بالبيانات اللازمة، وعند الوصول إلى View نقوم بتعريف عنصر مثل ListBox, DataGrid والتي بإمكانها عرض مجموعة من العناصر، ويتم الربط عن طريق الخاصية ItemsSource.

إنشاء PeopleViewModel

1. نقوم بإضافة كلاس جديد إلى مجلد ViewModel ونعطيه إسم PeopleViewModel، ثم نضيف خاصية (Property) جديدة من نوع List إلى الكلاس الجديد! لكن السؤال هنا، ما هو نوع البيانات الذي ستحويه هذه المجموعة؟

   قد يخطر ببالك أننا نتعامل مع مجموعة من الأشخاص(Person) لذلك سيكون شكلها

   List<Person> People;

   بالطبع هذا ممكن، لكن مع MVVM فإن تكلفة هذا العمل مرتفعة! ما الحل إذاً؟ الحل بما أننا أنشأنا سابقاً كلاس PersonViewModel بإمكاننا استغلاله بشكل أكبر(وهذا ما سنراه لاحقاً)، وكما نعلم أن PersonViewModel تحوي جميع البيانات المطلوبة لـِ Person معين، لذلك يمكن استخداماها كنوع بيانات لتلك المجموعة بكل سهولة.

   الآن بعد أن تعرفنا على ما يجب إضافته نقوم بعمل كلاس PeopleViewModel كالتالي:

```
namespace Mvvm.Session01.ViewModel
{
    using Mvvm.Session01.Model;
    using System.Collections.Generic;
    public class PeopleViewModel
    {
        public List<PersonViewModel> People { get; set; }

        public PeopleViewModel()
        {
            People = new List<PersonViewModel>()
            {
                new PersonViewModel(new Person(){
                    Id = 1, FirstName="Tareq", LastName="Jehad"}),
```

```
                new PersonViewModel(new Person(){
                    Id = 2, FirstName="Ahmad", LastName="Mohammad"}),

                new PersonViewModel(new Person(){
                    Id = 3, FirstName="Visual",LastName="Studio"}),

                new PersonViewModel(new Person(){
                    Id = 4, FirstName="MVVM", LastName="Way"})
            };
        }
    }
}
```

لاحظ أننا قمنا بتعبئة البيانات مباشرة ضمن الـ Constructor، بالطبع هذا العمل مبدئياً كوننا لا نتعامل مع قواعد بيانات، والامر مشابه مع قواعد البيانات، حيث سنقوم بطلب تلك البيانات من مزود معين للبيانات، كما سنلاحظ لاحقاً.

2. نقوم بإضافة عنصر جديد من نوع UserControl ونعطيه إسم People.xaml ونضعه ضمن مجلد الـ View.

الآن نقوم بإضافة عنصر DataGrid إلى People.xaml لنعرض مجموعة البيانات عليه، لذلك عدل كود XAML كالتالي:

```
<UserControl x:Class="Mvvm.Session01.View.People"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:ViewModels="clr-namespace:Mvvm.Session01.ViewModel">
    <UserControl.DataContext>
        <ViewModels:PeopleViewModel/>
    </UserControl.DataContext>
    <Grid>
        <DataGrid ItemsSource="{Binding People}"
                  AutoGenerateColumns="False"
                  CanUserAddRows="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding Id}"/>
                <DataGridTextColumn Header="First Name" Binding="{Binding FirstName}"/>
                <DataGridTextColumn Header="Last Name" Binding="{Binding LastName}"/>
                <DataGridTextColumn Header="Full Name" Binding="{Binding FullName}"/>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</UserControl>
```

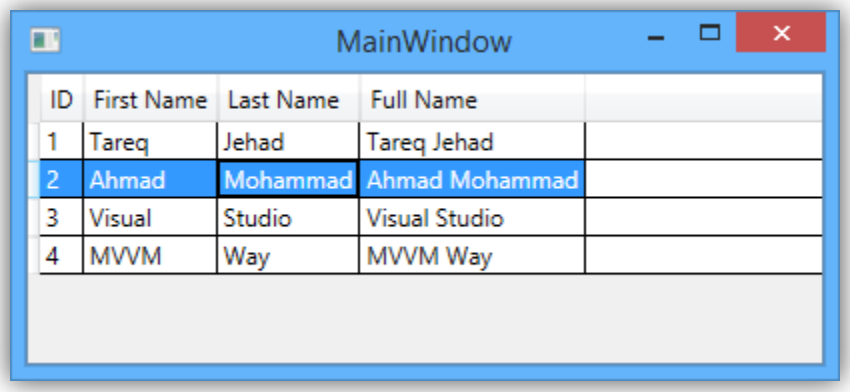
3. مرحلة ربط View مع ViewModel كما تحدثنا سابقاً تقوم عن طريق ربط خاصية DataContext مع اوبجكت من ViewModel، وقد تمت سابقاً عن طريق كود الواجهة، الآن للتعرف على الطريقة

الجديدة، تقوم بالكامل عن طريق XAML، حيث نقوم بتعريف فضاء الأسماء للإشارة إلى مجلد ViewModel، ثم نقوم بتعريف اوبجكت جديد ضمن UserControl.DataContext

4. الآن ولتجربة العمل قم بالتعديل على MainWindow.xaml، فبدل أن تعرف عليها عنصر من نوع Person عرفه People، كالتالي:

```
<Window x:Class="Mvvm.Session01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:Views="clr-namespace:Mvvm.Session01.View"
        Title="MainWindow" Height="140" Width="340">
    <Grid>
        <Views:People/>
    </Grid>
</Window>
```

5. قم بتشغيل التطبيقF5 .



السؤال المطروح الآن، كيف يتم التعامل والتفاعل مع هذه الواجهة العمياء؟

كما قلنا سنبدأ بمناقشة المشاكل للحصول على الحل المناسب. لذلك سنبدأ الآن بأبسط تفاعل وهو حذف عنصر من هذه المجموعة، ثم ننتقل لمرحلة الإضافة والحذف.

تواصل الـ View مع الـ ViewModel

بالطبع يجب تطبيق عمليات الحذف والإضافة والتعديل ضمن ViewModel، لذلك لابد من معرفة العنصر المحدد على View، مثلاً على الشكل السابق قام المستخدم بتحديد العنصر الثاني، لكن الـ ViewModel لا تملك أدنى فكرة عن ذلك؟ كيف يتم هذا التواصل إذاً؟

يتم التواصل أيضاً عن طريق DataBinding، أي نحن بحاجة لخاصية معينة ضمن ViewModel، بحيث نربطها مع View، ونحدد من خلالها السطر المحدد. وبما أن نوع البيانات ضمن السطر المحدد من نوع

PersonViewModel (كون البيانات كلها عبارة عن List<PersonViewModel>)، لذلك ستكون خاصية ViewModel من هذا النوع، لذلك قم بإضافة السطر التالي إلى PeopleViewModel

```
public PersonViewModel SelectedPerson { get; set; }
```

أصبح الآن بإمكاننا الذهاب إلى View وربط هذه الخاصية مع خاصية مشابه ضمن DataGrid تسمى SelectedItem، لذلك قم بتعديل Peroson.xaml بإضافة السطر الجديد:

```
...
<DataGrid ItemsSource="{Binding People}"
                    SelectedItem="{Binding SelectedPerson}"
                    AutoGenerateColumns="False"
...
```

لنرجع إلى PeopleViewModel ونقوم بإجراء التعديلات البسيطة التالية:

1. قم بإضافة ميثود جديدة Delete لحذف عنصر معين، بالشكل التالي:

```
private bool CanDelete()
{
    return (SelectedPerson != null);
}

public void Delete()
{
    if (CanDelete())
        People.Remove(SelectedPerson);
}
```

2. الآن أصبح بإمكاننا حذف عنصر بشرط أن يكون المستخدم قد اختاره مسبقاً، لذلك نحن فقط بحاجة لطب هذه ميثود الحذف. لذلك قم بتعديل People.xaml، لإضافة زر للحذف، كالتالي:

```
<UserControl x:Class="Mvvm.Session01.View.People"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:ViewModels="clr-namespace:Mvvm.Session01.ViewModel">
    <UserControl.DataContext>
        <ViewModels:PeopleViewModel/>
    </UserControl.DataContext>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <DataGrid  ItemsSource="{Binding People}"
                   SelectedItem="{Binding SelectedPerson}"
                   AutoGenerateColumns="False"
```

```
                    CanUserAddRows="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="ID" Binding="{Binding Id}"/>
                <DataGridTextColumn Header="First Name" Binding="{Binding
FirstName}"/>
                <DataGridTextColumn Header="Last Name" Binding="{Binding LastName}"/>
                <DataGridTextColumn Header="Full Name" Binding="{Binding FullName}"/>
            </DataGrid.Columns>
        </DataGrid>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Button Content="Delete" Click="Button_Click"/>
        </StackPanel>
    </Grid>
</UserControl>
```

كما تحدثنا سابقاً يجب ربط View مع ViewModel عن طريق DataBinding، بالتالي لسنا بحاجة إلى CodeBehind، كما سنكتب هنا، لكني أفضّل أولاً إظهار المشكلة التي نتج عنها ميزة جديدة، كما سنشاهد أن Commands ظهرت لربط أحداث الـ View مع ViewModel، أما الآن سنعالج الحدث بالطريقة التقليدية.

3. ضمن حدث الزر، يجب علينا طلب ميثود الحذف من PeopleViewModel المربوط مع هذه الـ View، كالتالي:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
        PeopleViewModel peopleViewModel = (PeopleViewModel)DataContext;
        peopleViewModel.Delete();
}
```

4. الآن قم بتشغيل البرنامج، فتظهر الشاشة كما في الشكل التالي، حدد عنصر معين ثم قم بالضغط على حذف! ستلاحظ أن عملية الحذف لا تتم على View كما هو مطلوب منها، مع أننا قمنا بحذف ذلك العنصر من People List، السبب كنا قد ذكرناه سابقاً، وقتها لم تقم View بتحديث قيمة FullName!

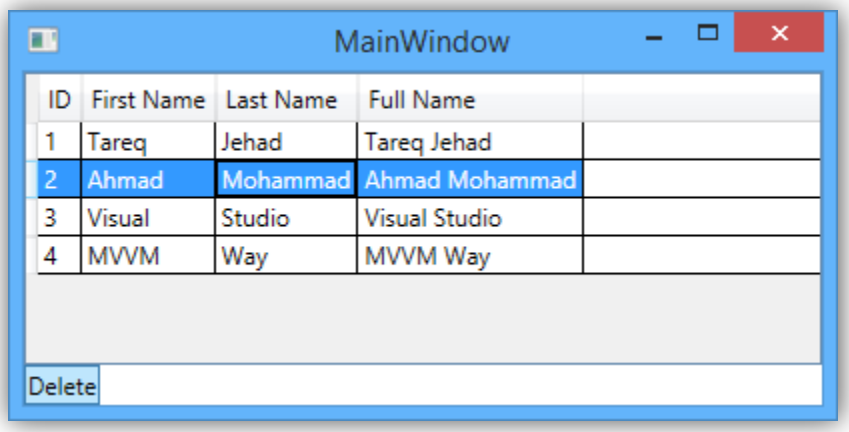
إجابتك ربما ستكون، إننا بحاجة لتنبيه الـ View عند حذف عنصر معين، لتحديث بياناتها، الجواب نعم.

لكن ليس بالطريقة السابقة، لأنها لا تنفع هنا، كون لدينا مجموعة من العناصر وليس عنصر واحد.

لذلك قدمت WPF مفهوم جميل يقوم بكل هذا العمل، هو ObservableCollection حيث نقوم بإستخدامه مثل استخدام List, ArrayList العادية، لكنها تملك القدر على تنبيه العناصر المرتبطة معها، لذلك عدل تعريف People ضمن PeopleViewModel لتصبح كالتالي:

```
public ObservableCollection<PersonViewModel> People { get; set; }
```

تقع هذه المجموعة تحت فضاء الأسماء System.Collections.ObjectModel لذلك لا تنسى إضافته.



بعد أن شاهدنا العمليات السابقة، نأتي إلى مرحلة متقدمة نوعاً ما وهي إمكانية إضافة عناصر جديدة أو تعديل عناصر سابقة، لكن هذه العمليات تتطلب مجموعة من المهام يمكن إنجازها عن طريق الأحداث (Events)، بين الواجهة View والـ ViewModel، وكما أشرنا سابقاً من أن كتابة هذه الأحداث ضمن كود الواجهة مكلف، لذلك سننتقل بدايةً لمفهوم ربط تلك الأحداث عن طريق Commands، حيث سنقوم بربط زر الحذف مع DeleteCommand، ثم سنشاهد مجموعة من التحسينات على تطبيقنا الحالي، وبعدها يصبح بإمكاننا إضافة عمليات التعديل والإضافة بسهولة.

Commands

الـ Commands عبارة عن نمط تصميم Command Pattern يعتمد على التواصل بين طبقتين منفصلتين، بنفس آلية الأحداث، لذلك قامت ميكروسوفت بتطبيق هذا النمط ضمن XAML وأصبح التعامل معه سهل جداً كما سنرى.

ما هي عناصر الـ Command الأساسية:

لتعريف Command معين يلزمنا عمل كلاس جديد يرث من ICommand Interface والذي سيزودنا بثلاث عناصر وهي:

1. Execute() ميثود، لتنفيذ المهمة المطلوبة.
2. CanExecute() للتحقق من إمكانية تنفيذ الإمكانية من عدمها.
3. CanExecuteChanged حدث، يجب رفعه أو تشغيله عند حدوث تعديل على القيم التي تحدد إمكانية تنفيذ المهمة (CanExecute).

كتابة DeleteCommand

1. قم بتعريف كلاس جديد، تحت كلاس PeopleViewModel وسميه DeleteCommand، ثم اجعله يرث من ICommand الموجود ضمن فضاء الأسماء System.Windows.Input، ثم قم بتطبيق محتوى هذا الـ Interface، ليصبح لديك شكل الكلاس كالتالي:

```
internal class DeleteCommand : ICommand
{
       public bool CanExecute(object parameter)
       {
          throw new System.NotImplementedException();
       }

       public event System.EventHandler CanExecuteChanged;

       public void Execute(object parameter)
       {
           throw new System.NotImplementedException();
       }
}
```

2. لكي يتم التعرف على ViewModel وتنفيذ مهامه ضمن هذا Command، يجب أخذ Reference منه، ثم العمل معه، لذلك قم بتعريف _ViewModel PeopleViewModel في كلاس DeleteCommand، ثم اضف Constructor بحيث نمرر معه PeopleViewModel،

```
private PeopleViewModel _viewModel;
public DeleteCommand(PeopleViewModel viewModel)
{
     _viewModel = viewModel;
}
```

3. الآن أصبح بالإمكان التعامل مع PeopleViewModel، وبنفس الطريقة التي كتبنا بها ميثود الحذف سابقاً سنعيد كتابتها هنا، لذلك عدل على Execute, CanExeucte كالتالي:

```
public bool CanExecute(object parameter)
{
        return _viewModel.SelectedPerson != null;
}
public void Execute(object parameter)
{
        _viewModel.People.Remove(_viewModel.SelectedPerson);
}
```

4. يبقى لدينا الحدث CanExeucteChanged، وهو مهم جداً في هذه العملية والسبب المهم كالتالي:

عندما يتم ربط Command معين مع عنصر على View، فإن ذلك العنصر يكون ذكي كفاية ليتعرف على إمكانية تنفيذ تلك الـ Command ام لا،فمثلاً الزر يعطل نفسه عند عدم إمكانية التنفيذ، في مثالنا عندما يكون SelectedPerson = null، أما في اللحظة التي يحدد المستخدم سطر معين، فإن الزر تلقائيا سيستجب لذلك التغيير ويفعل نفسه، إذناً منه بتنفيذ عملية الحذف، وما الحدث CanExeucteChanged إلا آليه لتنبيه ذلك الزر عن تغير قيمة SelectedPerson.

طبعاً هناك أكثر من طريقة لتطبيق هذا الحدث، سنكتب أفضل طريقة هنا، حيث نقوم بتسجيل هذا الحدث ضمن CommandManager، وعند إنتهاءه سيقوم CommandManager بحذفه أيضاً، لكي لا يحصل ضياع في الذاكرة Memory Leak. بالتالي سيكون شكل DeleteCommand النهائي كالتالي:

```
public class DeleteCommand : ICommand
{
   private PeopleViewModel _viewModel;
   public DeleteCommand(PeopleViewModel viewModel)
   {
       _viewModel = viewModel;
   }
   public bool CanExecute(object parameter)
   {
      return _viewModel.SelectedPerson != null;
   }
   public void Execute(object parameter)
   {
       _viewModel.People.Remove(_viewModel.SelectedPerson);
   }

   public event System.EventHandler CanExecuteChanged
   {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
   }
}
```

جميل! الآن وكما تعودنا لكي تصبح هذه الـ Command متاحة ل View يجب عرضها كخاصية ضمن ViewModel لذلك لنعرف DeleteCommand Property ضمن PeopleViewModel كالتالي:

```
public ICommand DeleteCommand { get; set; }
```

نحن بحاجة لتهيئتها بالطبع، ويتم ذلك ضمن PeopleViewModel Constructor كالتالي:

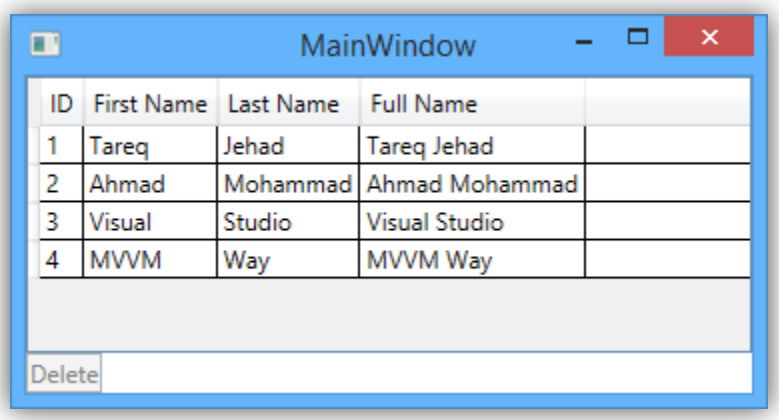
```
DeleteCommand = new DeleteCommand(this);
```

5. الآن بقي أن نربطها مع الـ View، لذلك نذهب إلى People.xaml إلى زر الحذف ونعدله كالتالي:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
      <Button Content="Delete" Command="{Binding DeleteCommand}"/>
</StackPanel>
```

ولا ننسى أن نمسح الحدث Click، فلم نعد نحتاجه.

بعد تشغيل البرنامج وقبل أي شيء، نلاحظ أن الزر Delete غير مفعل، (إذا لم تلاحظ ذلك عندك، راجع الخطوات السابقة)، بمجرد تحديد سطر سيتم تفعيل الزر مباشرة.



الآن وقبل الإنتقال لعمليات الإضافة والتعديل، ولكي لا نعيد كتابة كلاسات جديدة في كل مرة، سنقوم بكتابة كلاس عام نسميه MVVMCommand مثلاً، ومن خلاله نستطيع تعريف أي Command أخرى مباشرة دون الحاجة لكتابة كلاسات جديدة في كل مرة.

مع المرور أكثر على ميزات MVVM سنتعرف على الكثير من المفاهيم الجديدة، والحلول الرائعة التي اتيحت لها، كما سنكتب في كلاس MVVMCommand العام، وسنقوم لاحقاً أيضاًبإنشاء كلاسات جديدة وعامة تفيدنا كثيراً في اختصار الوقت والجهد. كأن نعرف ViewModel Base Class، وغيرها،،،

كل هذا نتج عنه مجموعة من الأدوات الجاهزة Frameworks تحوي على الكثير من الكلاسات الجاهزة والأدوات الرائعة لتطوير تطبيق MVVM بسهولة، فليس من الضروري البدء بكتابة التطبيق من الصفر، ونحن هنا نعرف المفاهيم من البداية لكي لا يكون هناك غموض، وبنفس الوقت نكون قد كتبنا Framework خاص بنا.

كتابة كلاس MVVMCommand العام

1. قم بإضافة مجلد جديد للمشروع وسميه Command، ثم قم بإضافة كلاس جديد إليه تحت إسم MVVMCommand، واجلعه يرث من ICommand كما فعلنا سابقاً.

```
namespace Mvvm.Session01.Command
{
    using System;
    using System.Windows.Input;
    public class MVVMCommand : ICommand
    {
        public bool CanExecute(object parameter)
        {
            throw new NotImplementedException();
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
        public void Execute(object parameter)
        {
            throw new NotImplementedException();
        }
    }
}
```

2. ما هي المهام التي سيقوم هذا الكلاس بتنفيذها، واي ViewModel سيأخذ؟ بالطبع هدفنا هو اختصار تلك الطريقة الطويلة، لذلك ما سنقوم به، هو تعريف مؤشرات على ميثودز Delegates ، ولاحقاً عندما نقوم بتعريف Command معينة، نقوم بإسناد تلك الميثودز أو الدوال، لذلك سنقوم بتعريف مؤشر على Exectute واخر على CanExecute كالتالي:

```
private readonly Action _execute;
private readonly Func<bool> _canExecute;
```

```
public MVVMCommand(Action execute, Func<bool> canExecute)
{
    if (execute == null)
        throw new ArgumentException("Execute");

    _execute = execute;
    _canExecute = canExecute;
}

public MVVMCommand(Action execute)
  : this(execute, null)
{ }
```

هنا قمنا بتعريف تلك المؤشرات وإسنادها ضمن Constructor، بالطبع يلزمنا أن تكون Action ذات قيمة، فهي ميثود التنفيذ، اما ميثود إمكانية التنفيذ CanExecute، فهي لا تلزمنا دائما، مثلاً إضافة سطر جديد، ليس عليه شرط معين، أما الحذف والتعديل تحتاج ذلك الشرط، لذلك وضعنا Constructor الثاني.

3. الآن عند تنفيذ Command فإن الميثود Exeucte ضمن هذا الكلاس هي التي ستتنفذ، لذلك سنسند إليها المؤشر _exeucte ليتنفذ، والحال نفسه مع CanExecute، أما بالنسبة للحدث CanExecuteChanged فنحن بحاجة لكتابته كما كتبناه سابقاً، بالإضافة لشرط جديد، وهو أن يكون هناك مؤشر CanExecute، وليس null، كما شرحنا آنفاً.
4. الشكل النهائي لهذا الكلاس سيكون كالتالي:

```
namespace Mvvm.Session01.Command
{
    using System;
    using System.Windows.Input;
    public class MVVMCommand : ICommand
    {
        private readonly Action _execute;
        private readonly Func<bool> _canExecute;

        public MVVMCommand(Action execute, Func<bool> canExecute)
        {
            if (execute == null)
                throw new ArgumentException("Execute");

            _execute = execute;
            _canExecute = canExecute;
        }

        public MVVMCommand(Action execute)
            : this(execute, null)
        { }

        public bool CanExecute(object parameter)
```

```
        {
            if (_canExecute == null)
                return true;
            return _canExecute();
        }

        public event EventHandler CanExecuteChanged
        {
            add
            {
                if (_canExecute != null)
                    CommandManager.RequerySuggested += value;
            }
            remove
            {
                if (_canExecute != null)
                    CommandManager.RequerySuggested -= value;
            }
        }

        public void Execute(object parameter)
        {
            _execute();
        }
    }
}
```

5. هكذا أصبح تعريف Commands أسهل، وهو ما سيوفر علينا الجهد لاحقاً، لذلك سنعيد كتابة DeleteCommand من جديد لنرى الفائدة، كل ما نحتاج إليه هو الذهاب إلى

PeopleViewModel وبدل أن نهيء DeleteCommand من كلاس DeleteCommand، سنقوم بالتهيئة من MVVMCommand، وسنستفيد من الميثودز التي كنا قد كتبناها سابقاً Delete() و CanDelete، لذلك التعديل سيتم ضمن PeopleViewModel Constructor كالتالي:

```
DeleteCommand = new MVVMCommand(Delete, CanDelete);
```

ولا تنسى أضافة فضاء الأسماء للمجلد Command ضمن المشروع.

6. نفذ التطبيق، وشاهد نفس النتيجة، ولم نعد بحاجة بالطبع للكلاس DeleteCommand.

إضافة عنصر جديد إلى People

سنقوم الأن بإضافة عنصر جديد إلى People، واتمنى لو تطبق الفكرة بنفسك، إعتماداً على ما تعلمناه.

1. نقوم بتعريف Command جديدة ضمن PeopleViewModel بإسم NewCommand، من نوع MVVMCommand، كما فعلنا مع DeleteCommand

```
public ICommand NewCommand { get; set; }
```

2. هنا لسنا بحاجة سوى لميثود واحد لإضافة عنصر جديد من نوع PersonViewModel إلى People، لذلك عرف ميثود جديدة بإسم New، وإضف خلالها ذلك العنصر، كالتالي:

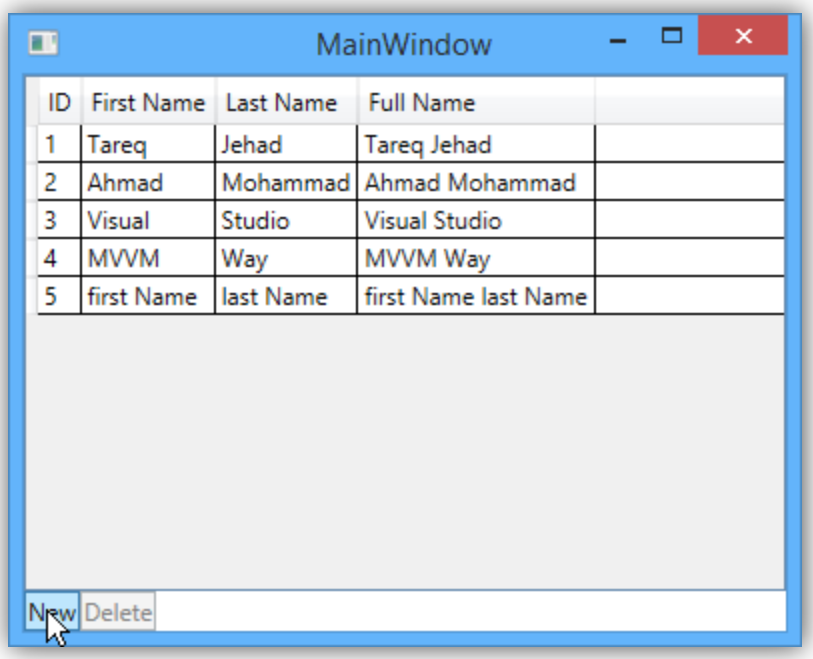
```
private int id = 5;
private void New()
{
    Person person = new Person
     {
         Id = id,
         FirstName= "first Name",
         LastName = "last Name"
      };

    People.Add(new PersonViewModel(person));
}
```

3. الآن قم بتهيئة هذه الـ Command ضمن الـ Constructor كالتالي:

```
NewCommand = new MVVMCommand(New);
```

4. إذهب إلى People.xaml وأضف زر جديد بجانب زر الحذف وضع الخاصية Content = New كالتالي:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
      <Button Content="New" Command="{Binding NewCommand}"/>
      <Button Content="Delete" Command="{Binding DeleteCommand}"/>
</StackPanel>
```



في المرة القادمة بإذن الله، سنقوم بإضافة العنصر الجديد عن طريق واجهة PersonView، حيث سنتمكن من ادخال القيم يدوياً، ومن ثم سنستخدم نفس الـ View للتعديل على سطر محدد.

حاول كتابة إحدى تلك المهام، او مهمة اخرى بنفس الطريقة التي عملنا بها.

Tareq Jehad
tareqjehad@yahoo.com
© 2014