**Задание. Разработка проекта Xamarin.Forms на основе паттерна MVVM (Model-View-ViewModel)**

# Индивидуальное задание

Создать и отладить проект Xamarin.Forms на основе паттерна MVVM для реализации следующих задач: отображение списка элементов и характеристик, переход на страницу описания выбранного элемента и всех его характеристик, создание нового элемента и ввод характеристик в отдельной странице, хранение элементов списка во временном файловом хранилище, отображение страницы «О приложении».

В зависимости от типа элемента, необходимо создать класс с уникальным набором характеристик.

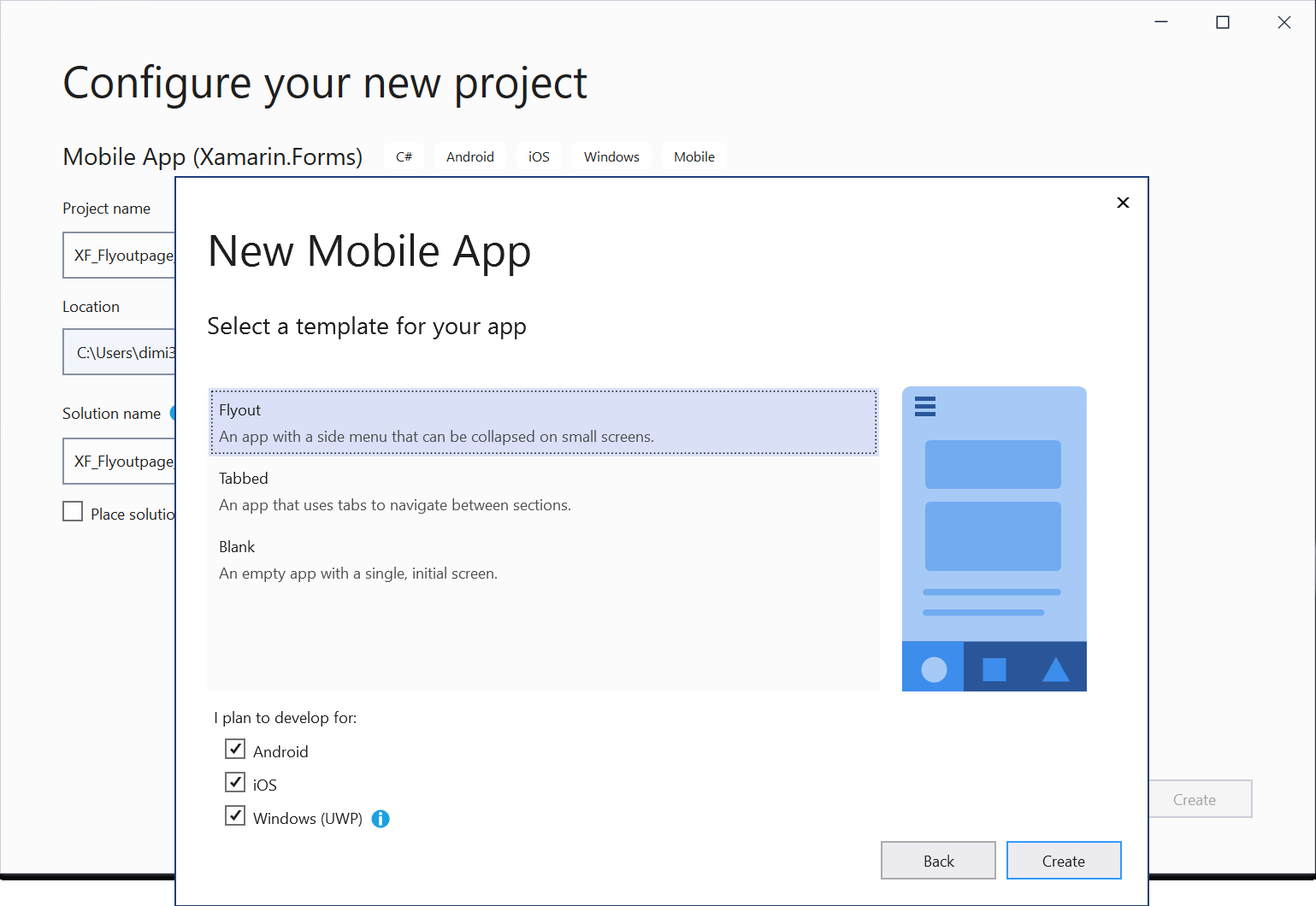
Элементы и характеристики по вариантам:

|  |  |  |
| --- | --- | --- |
| Вариант | Элементы | Количество различных характеристик элемента |
| 1 | Автомобили | 5 |
| 2 | Планшеты | 8 |
| 3 | Студенты | 6 |
| 4 | Университеты | 7 |
| 5 | IT корпорации | 5 |
| 6 | Аксессуары смартфонов | 7 |
| 7 | Товары для компьютеров | 5 |
| 8 | Продукты питания | 7 |
| 9 | Моноблоки | 5 |
| 10 | Мебель | 8 |
| 11 | Одежда | 5 |
| 12 | Аксессуары для автомобилей | 6 |
| 13 | Овощная продукция | 5 |
| 14 | Цветы | 6 |
| 15 | Программное обеспечение | 8 |
| 16 | Запчасти автомобилей | 5 |
| 17 | Электроинструмент | 7 |
| 18 | Офисное оборудование | 8 |
| 19 | Канцелярская продукция | 6 |
| 20 | Книжная продукция | 5 |

Важнейшая цель проекта – продемонстрировать возможности применения паттерна проектирования MVVM в рамках проекта мобильного приложения Xamarin.Forms.

# Теория и практика

Один из подходящих типов проекта Visual Studio для выполнения задания:



Паттерн MVVM (Model - View - ViewModel) основывается на разделении функциональной части приложения на три ключевых компонента:

* **View** - представление или пользовательский интерфейс
* **Model** - модель или данные, которые используются в приложении
* **ViewModel** - промежуточный слой между представлением и данными, который обеспечивает их взаимодействие

Преимуществом использования данного паттерна является меньшая связанность между компонентами и разделение ответственности между ними. То есть Model отвечает за данные, View отвечает за графический интерфейс, а ViewModel - за логику приложения.

Легкость реализации паттерна MVVM в Xamarin Forms стала возможной благодаря ранее рассмотренному механизму привязки.

Рассмотрим простейший пример. Определим класс данных или модели:

|  |
| --- |
| public class Phone  {      public string Title { get; set; }      public string Company { get; set; }      public int Price { get; set; }  } |

Также добавим в проект класс, который назовем PhoneViewModel со следующим содержимым:

|  |
| --- |
| using System.ComponentModel;  public class PhoneViewModel :INotifyPropertyChanged  {      public event PropertyChangedEventHandler PropertyChanged;      private Phone phone;        public PhoneViewModel()      {          phone = new Phone();      }      public string Title      {          get { return phone.Title; }          set          {              if (phone.Title != value)              {                  phone.Title = value;                  OnPropertyChanged("Title");              }          }      }      public string Company      {          get { return phone.Company; }          set          {              if (phone.Company != value)              {                  phone.Company = value;                  OnPropertyChanged("Company");              }          }      }      public int Price      {          get { return phone.Price; }          set          {              if (phone.Price != value)              {                  phone.Price = value;                  OnPropertyChanged("Price");              }          }      }      protected void OnPropertyChanged(string propName)      {          if (PropertyChanged != null)              PropertyChanged(this, new PropertyChangedEventArgs(propName));      }  } |

Это и будет компонент ViewModel, который связывает данные и визуальный интерфейс. По большому счету, она представляет обертку над классом Phone, определяя все те же свойства. Для упрощения задачи сам объект Phone создается в конструкторе, хотя в реальности там могла бы быть более сложная логика, например, по получению объекта из базы данных.

Важно, что данный класс реализует интерфейс INotifyPropertyChanged, что позволяет уведомлять систему об изменении его свойств с помощью события PropertyChanged.

Теперь создадим визуальную часть. Определим на главной странице MainPage.xaml следующее содержимое:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <ContentPage xmlns="<http://xamarin.com/schemas/2014/forms>"               xmlns:x="<http://schemas.microsoft.com/winfx/2009/xaml>"               x:Class="MvvmApplication.MainPage">    <StackLayout>      <Label Text="{Binding Title}" FontSize="Medium" />      <Label Text="{Binding Company}" FontSize="Medium" />      <Label Text="{Binding Price}" FontSize="Medium" />    </StackLayout>  </ContentPage> |

Здесь определена привязка к свойству ViewModel.

А в конструкторе страницы в файле кода c# пропишем в качестве контекста данных для страницы определенную ранее ViewModel:

|  |
| --- |
| public partial class MainPage : ContentPage  {      public MainPage()      {          InitializeComponent();          this.BindingContext = new PhoneViewModel          {              Title = "iPhone 7",              Company = "Apple",              Price=52000          };      }  } |

И при запуске приложение выведет нам все данные о viewmodel, переданной во view:

При использовании паттерна в качестве представления обычно выступает страница, и, как правило, одно представление (одна страница) связана с одной моделью представления (ViewModel). При этом одну и ту же VıewModel могут использовать несколько представлений.

Стоит отметить, что в этом отношении ViewModel ничего не знает о представлении, что это, какие элементы управления оно содержит. ViewModel только определяет логику обработку без какой-либо связи с графическим интерфейсом.

# Команды и взаимодействие с пользователем в MVVM

Как правило, необходимо не только отображать данные, но и использовать какую-то логику взаимодействия с пользователем, обрабатывать пользовательский ввод. Рассмотрим, как это сделать в рамках паттерна MVVM.

Ключевой идеей паттерна является взаимодействие с моделью через ViewModel, то есть в данном случае использование событий визуальных компонентов и их обработчиков нежелательно. Чтобы решить эту задачу, в платформе Xamarin Forms имеется механизм команд.

Механизм команд раскрывается через интерфейс ICommand:

public interface ICommand

{

void Execute(object arg);

bool CanExecute(object arg);

event EventHandler CanExecuteChanged;

}

Интерфейс ICommand определен в пространстве имен System.Windows.Input и расположен в сборке System.ObjectModel.

Встроенную поддержку команд имеют следующие элементы управления: Button, MenuItem, ToolbarItem, SearchBar, TextCell, ImageCell, ListView, TapGestureRecognizer.

Итак, для применения команд возьмем проект приложения из прошлой темы и определим в нем простейшую команду. Для этого добавим в проект класс IncreasePriceCommand:

using System;

using System.Windows.Input;

namespace MvvmApplication

{

public class IncreasePriceCommand : ICommand

{

public event EventHandler CanExecuteChanged;

PhoneViewModel viewModel;

public IncreasePriceCommand(PhoneViewModel vm)

{

viewModel = vm;

}

public bool CanExecute(object parameter)

{

return viewModel.Price < 100000;

}

public void Execute(object parameter)

{

if (CanExecute(parameter))

viewModel.IncreasePrice();

}

}

}

Команда через конструктор принимает объект PhoneViewModel. Метод CanExecute возвращает true, если у цена во viewModel меньше 100000. Здесь можно определить любое условие. В прицнипе данный метод вообще не обязательно использовать.

Метод Execute() собственно выполняет действие команды. В нашем случае будет вызываться метод IncreasePrice, который мы определим в ViewModel. Стоит отметить, что метод принимает параметр типа object, то есть мы можем извне передавать в команду различные данные.

Теперь изменим класс PhoneViewModel, чтобы он использовал данную команду:

using System.ComponentModel;

using System.Windows.Input;

namespace MvvmApplication

{

public class PhoneViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

private Phone phone;

public ICommand Increase { get; }

public PhoneViewModel()

{

phone = new Phone();

Increase = new IncreasePriceCommand(this);

}

public void IncreasePrice()

{

if (phone != null)

Price += 100;

}

public string Title

{

get { return phone.Title; }

set

{

if (phone.Title != value)

{

phone.Title = value;

OnPropertyChanged("Title");

}

}

}

public string Company

{

get { return phone.Company; }

set

{

if (phone.Company != value)

{

phone.Company = value;

OnPropertyChanged("Company");

}

}

}

public int Price

{

get { return phone.Price; }

set

{

if (phone.Price != value)

{

phone.Price = value;

OnPropertyChanged("Price");

}

}

}

protected void OnPropertyChanged(string propName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

ViewModel содержит команду Increase, которая представляет тип IncreasePriceCommand.

И также изменим код страницы в xaml, добавив в него кнопки с привязкой к вышеопределенным командам:

<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="MvvmApplication.MainPage">

<StackLayout>

<Label Text="{Binding Title}" FontSize="Medium" />

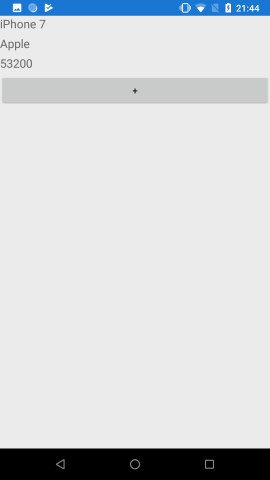
<Label Text="{Binding Company}" FontSize="Medium" />

<Label Text="{Binding Price}" FontSize="Medium" />

<Button Text="+" Command="{Binding Increase}" />

</StackLayout>

</ContentPage>



С помощью свойства Command и выражения привязки кнопка связывается с командой Increase. И при нажатии на кнопку происходит действие, заложенное в связанной команде, то есть произойдет увеличение цены товара.

Также стоит отметить, что если мы хотим использовать команды, то нам необязательно определять новый класс команды. В Xamarin.Forms есть встроенный класс Command, который реализует всю функциональность. Нам же достаточно передать в его конструктор нужно действие, которое соответствует делегату Action. Например, изменим класс PhoneViewModel следующим образом:

using System.ComponentModel;

using System.Windows.Input;

using Xamarin.Forms;

namespace MvvmApplication

{

public class PhoneViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

private Phone phone;

public ICommand Increase { get; }

public PhoneViewModel()

{

phone = new Phone();

Increase = new Command(IncreasePrice);

}

public void IncreasePrice()

{

if (phone != null)

Price += 100;

}

public string Title

{

get { return phone.Title; }

set

{

if (phone.Title != value)

{

phone.Title = value;

OnPropertyChanged("Title");

}

}

}

public string Company

{

get { return phone.Company; }

set

{

if (phone.Company != value)

{

phone.Company = value;

OnPropertyChanged("Company");

}

}

}

public int Price

{

get { return phone.Price; }

set

{

if (phone.Price != value)

{

phone.Price = value;

OnPropertyChanged("Price");

}

}

}

protected void OnPropertyChanged(string propName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

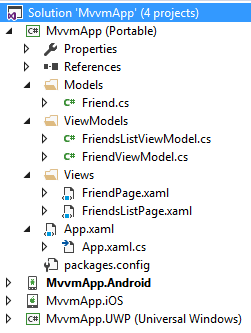
}

}

При запуске проекта приложение будет работать также, как и на предыдущем скриншоте.

# Другой пример проекта MVVM Xamarin.Forms

Создадим новое приложение, которое назовем MvvmApp. Задача нашего приложения будет заключаться в создании списка друзей, а также их добавлении, редактировании и удалении.

Вначале добавим в главный проект три новых папки для каждого из компонентов паттерна:

* Папка Models
* Папка ViewModels
* Папка Views

Страницу MainPage.xaml, если она имеется в проекте по умолчанию, можно удалить.

Финальная структура проекта будет выглядеть следующим образом:

Вначале определим в папке Models модель, которая будет представлять данные - класс Friend:

namespace MvvmApp.Models

{

public class Friend

{

public string Name { get; set; }

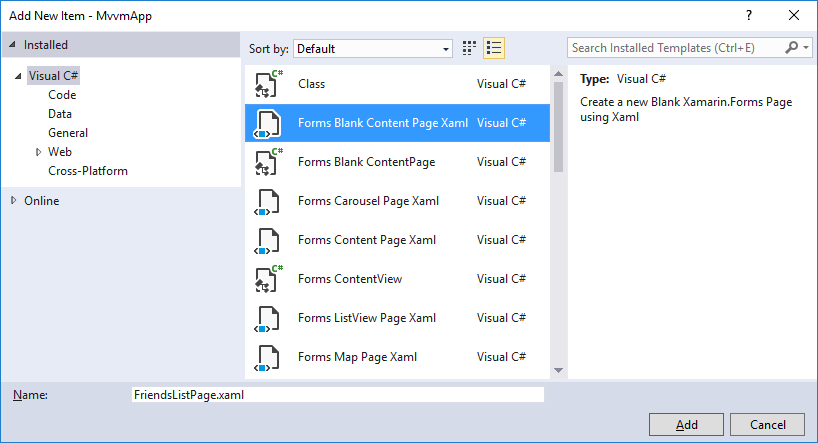
public string Email { get; set; }

public string Phone { get; set; }

}

}

Чтобы работать с данными этой модели добавим в папку Views две страницы XAML по типу ContentPage: FriendsListPage (для вывода списка друзей) и FriendPage (для управления одним другом).



Далее добавим в другую папку ViewModels класс FriendViewModel:

using System.ComponentModel;

using MvvmApp.Models;

namespace MvvmApp.ViewModels

{

public class FriendViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

FriendsListViewModel lvm;

public Friend Friend { get; private set; }

public FriendViewModel()

{

Friend = new Friend();

}

public FriendsListViewModel ListViewModel

{

get { return lvm; }

set

{

if (lvm != value)

{

lvm = value;

OnPropertyChanged("ListViewModel");

}

}

}

public string Name

{

get { return Friend.Name; }

set

{

if (Friend.Name != value)

{

Friend.Name = value;

OnPropertyChanged("Name");

}

}

}

public string Email

{

get { return Friend.Email; }

set

{

if(Friend.Email != value)

{

Friend.Email = value;

OnPropertyChanged("Email");

}

}

}

public string Phone

{

get { return Friend.Phone; }

set

{

if (Friend.Phone != value)

{

Friend.Phone = value;

OnPropertyChanged("Phone");

}

}

}

public bool IsValid

{

get

{

return ((!string.IsNullOrEmpty(Name)) ||

(!string.IsNullOrEmpty(Phone)) ||

(!string.IsNullOrEmpty(Email)));

}

}

protected void OnPropertyChanged(string propName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

Фактически этот класс является надстройкой над объектом Friend.

Затем добавим в папку ViewModels новый класс, который будет представлять список друзей и который будет использоваться на странице FriendsListPage - класс FriendsListViewModel:

using System.Collections.ObjectModel;

using System.Windows.Input;

using Xamarin.Forms;

using System.ComponentModel;

using MvvmApp.Views;

namespace MvvmApp.ViewModels

{

public class FriendsListViewModel : INotifyPropertyChanged

{

public ObservableCollection<FriendViewModel> Friends { get; set; }

public event PropertyChangedEventHandler PropertyChanged;

public ICommand CreateFriendCommand { protected set; get; }

public ICommand DeleteFriendCommand { protected set; get; }

public ICommand SaveFriendCommand { protected set; get; }

public ICommand BackCommand { protected set; get; }

FriendViewModel selectedFriend;

public INavigation Navigation { get; set;}

public FriendsListViewModel()

{

Friends = new ObservableCollection<FriendViewModel>();

CreateFriendCommand = new Command(CreateFriend);

DeleteFriendCommand = new Command(DeleteFriend);

SaveFriendCommand = new Command(SaveFriend);

BackCommand = new Command(Back);

}

public FriendViewModel SelectedFriend

{

get { return selectedFriend; }

set

{

if (selectedFriend != value)

{

FriendViewModel tempFriend = value;

selectedFriend = null;

OnPropertyChanged("SelectedFriend");

Navigation.PushAsync(new FriendPage(tempFriend));

}

}

}

protected void OnPropertyChanged(string propName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

private void CreateFriend()

{

Navigation.PushAsync(new FriendPage(new FriendViewModel() { ListViewModel = this }));

}

private void Back()

{

Navigation.PopAsync();

}

private void SaveFriend(object friendObject)

{

FriendViewModel friend = friendObject as FriendViewModel;

if (friend != null && friend.IsValid)

{

Friends.Add(friend);

}

Back();

}

private void DeleteFriend(object friendObject)

{

FriendViewModel friend = friendObject as FriendViewModel;

if (friend != null)

{

Friends.Remove(friend);

}

Back();

}

}

}

Для хранения списка друзей, который будет выводится на страницу, здесь определена коллекция Friends.

Навигация также является частью логики приложения, которая не относится к визуальной части, поэтому для хранения сервиса навигации здесь определено свойство Navigation. В дальнейшем через это свойство будет производиться переход к FriendPage.

Для управлением списком друзей в классе определено четыре команды. Команда добавления нового друга приводит в действие метод CreateFriend(), в котором производится переход к FriendPage. В конструктор FriendPage передается текущий объект FriendPageViewModel, который мы далее создадим.

По команде возвращения назад выполняется метод Back(), который производит переход назад.

Команда сохранения объекта выполняет метод SaveFriend(). В этом методе новый объект добавляется в коллекцию Friends.

По команде удаления вызывается метод DeleteFriend, который удаляет объект из списка.

Теперь изменим код страниц из папки Views. В коде C# страницы FriendsListPage пропишем следующее содержимое:

using Xamarin.Forms;

using MvvmApp.ViewModels;

namespace MvvmApp.Views

{

public partial class FriendsListPage : ContentPage

{

public FriendsListPage()

{

InitializeComponent();

BindingContext = new FriendsListViewModel() { Navigation = this.Navigation };

}

}

}

Здесь создается объект FriendsListViewModel, который устанавливается в качестве контекста страницы.

В коде XAML у этой страницы пропишем выражения привязки к этому объекту:

<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="MvvmApp.Views.FriendsListPage" Title="Список друзей">

<StackLayout>

<Button Text="Добавить" Command="{Binding CreateFriendCommand}" />

<ListView x:Name="booksList" ItemsSource="{Binding Friends}"

SelectedItem="{Binding SelectedFriend, Mode=TwoWay}" HasUnevenRows="True">

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<ViewCell.View>

<StackLayout>

<Label Text="{Binding Name}" FontSize="Medium" />

<Label Text="{Binding Email}" FontSize="Small" />

<Label Text="{Binding Phone}" FontSize="Small" />

</StackLayout>

</ViewCell.View>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage>

Несмотря на то, что тут определена кнопка для добавления нового объекта, код страницы не содержит никаких обработчиков, потому что всю обработку будет выполнять команда CreateFriendCommand, определенная в FriendsListViewModel.

Затем изменим код c# у страницы FriendPage:

using MvvmApp.ViewModels;

using Xamarin.Forms;

namespace MvvmApp.Views

{

public partial class FriendPage : ContentPage

{

public FriendViewModel ViewModel { get; private set; }

public FriendPage(FriendViewModel vm)

{

InitializeComponent();

ViewModel = vm;

this.BindingContext = ViewModel;

}

}

}

Теперь страница в качестве параметра в конструкторе принимает объект FriendViewModel, устанавливает его в качестве контекста и также не содержит никакой другой логики.

И в коде xaml у страницы определим выражения привязки:

<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="MvvmApp.Views.FriendPage" Title="Информация о друге">

<StackLayout>

<StackLayout x:Name="friendStack">

<Label Text="Имя" />

<Entry Text="{Binding Name}" FontSize="Medium" />

<Label Text="Электронная почта" />

<Entry Text="{Binding Email}" FontSize="Medium" />

<Label Text="Телефон" />

<Entry Text="{Binding Phone}" FontSize="Medium" />

</StackLayout>

<StackLayout Orientation="Horizontal" HorizontalOptions="CenterAndExpand">

<Button Text="Добавить" Command="{Binding ListViewModel.SaveFriendCommand}" CommandParameter="{Binding}" />

<Button Text="Удалить" Command="{Binding ListViewModel.DeleteFriendCommand}" CommandParameter="{Binding}" />

<Button Text="Назад" Command="{Binding Path=ListViewModel.BackCommand}" />

</StackLayout>

</StackLayout>

</ContentPage>

И при нажатии на кнопки будет вызываться соответствующая команда. Кроме того, первые две кнопки передают в команды параметр, который представляет привязанный объект - то есть тот объект FriendViewModel, который передан в конструктор и установлен в качестве контекста страницы. Выражение {Binding} без указания свойства объекта выполняет привязку к объекту, который является контекстом для данного элемента управления. Таким образом, команды в FriendsListViewModel получат добавляемый или удаляемый объект FriendViewModel.

При этому для редактирования в данном случае не надо нажимать никакую кнопку, так как при изменении значений в текстовых полях сразу же изменяться значения у этого объекта в списке. И после изменения значений достаточно нажать на кнопку Назад для возврата на главную страницу. Правда, такое поведение не всегда бывает удобно. В этом случае мы можем добавить к объекту Friend/FriendViewModel какой-нибудь идентификатор. А при редактировании передавать не объект из списка, а его копию. Затем при сохранении данных в зависимости от Id выполнять либо добавление (если Id не установлен), либо редактирование. Удаление в этом случае также бы производилось бы по Id.

И в конце в классе App установим страницу FriendsListPage в качестве главной:

using Xamarin.Forms;

using MvvmApp.Views;

namespace MvvmApp

{

public partial class App : Application

{

public App()

{

MainPage = new NavigationPage(new FriendsListPage());

}

protected override void OnStart()

{ }

protected override void OnSleep()

{ }

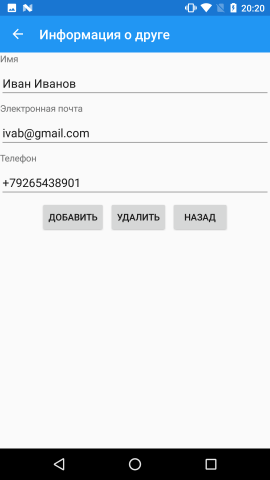
protected override void OnResume()

{ }

}

}

Запустим приложение и добавим какой-нибудь объект.



И после добавления мы увидим этот объект в списке на главной странице:



# Контекстное меню

Ключевым элементом для работы с наборами данных является ListView. И нередко бывает важно продумать, какие инструменты использовать для управления данными в ListView. Одним из таких инструментов является контекстное меню. Элемент ListView позволяет прикрепить к каждому элементу контекстное меню через свойство ContextActions.

Итак, для работы с контекстным меню создадим новый проект. В качестве объектов с которыми будем работать, возьмем класс Phone

public class Phone

{

public string Title { get; set; }

public string Company { get; set; }

public int Price { get; set; }

}

Далее добавим в проект модель PhoneViewModel, через которую мы будем взаимодействовать с объектами Phone:

using System.ComponentModel;

namespace MvvmApplication

{

public class PhoneViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

public Phone Phone { get; set; }

public PhonesListViewModel ListViewModel { get; set; }

public PhoneViewModel()

{

Phone = new Phone();

}

public string Title

{

get { return Phone.Title; }

set

{

if (Phone.Title != value)

{

Phone.Title = value;

OnPropertyChanged("Title");

}

}

}

public string Company

{

get { return Phone.Company; }

set

{

if (Phone.Company != value)

{

Phone.Company = value;

OnPropertyChanged("Company");

}

}

}

public int Price

{

get { return Phone.Price; }

set

{

if (Phone.Price != value)

{

Phone.Price = value;

OnPropertyChanged("Price");

}

}

}

protected void OnPropertyChanged(string propName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

Кроме свойств-надстроек над свойствами объекта Phone здесь есть свойство, которое представляет модель PhonesListViewModel - объект списка, в котором будет находиться текущий объект PhoneViewModel. И далее добавим в проект новый класс PhonesListViewModel:

using System.Collections.ObjectModel;

using System.Windows.Input;

using Xamarin.Forms;

namespace MvvmApplication

{

public class PhonesListViewModel

{

public ObservableCollection<PhoneViewModel> Phones { get; set; }

public ICommand MoveToTopCommand { protected set; get; }

public ICommand MoveToBottomCommand { protected set; get; }

public ICommand RemoveCommand { protected set; get; }

public PhonesListViewModel()

{

Phones = new ObservableCollection<PhoneViewModel>

{

new PhoneViewModel { Title="HP Elite z3", Price=55000, Company="HP", ListViewModel=this},

new PhoneViewModel {Title="Honor 8", Price= 28000, Company="Huawei", ListViewModel=this},

new PhoneViewModel {Title="iPhone SE", Price=30000, Company="Apple", ListViewModel=this },

new PhoneViewModel {Title="Galaxy Note 7", Price=60000, Company="Samsung", ListViewModel=this },

new PhoneViewModel {Title="Lumia 950 XL", Price=36000, Company="Microsoft", ListViewModel=this }

};

MoveToTopCommand = new Command(MoveToTop);

MoveToBottomCommand = new Command(MoveToBottom);

RemoveCommand = new Command(Remove);

}

private void MoveToTop(object phoneObj)

{

PhoneViewModel phone = phoneObj as PhoneViewModel;

if (phone == null) return;

int oldIndex = Phones.IndexOf(phone);

if (oldIndex > 0)

Phones.Move(oldIndex, oldIndex - 1);

}

private void MoveToBottom(object phoneObj)

{

PhoneViewModel phone = phoneObj as PhoneViewModel;

if (phone == null) return;

int oldIndex = Phones.IndexOf(phone);

if (oldIndex < Phones.Count-1)

Phones.Move(oldIndex, oldIndex + 1);

}

private void Remove(object phoneObj)

{

PhoneViewModel phone = phoneObj as PhoneViewModel;

if (phone == null) return;

Phones.Remove(phone);

}

}

}

Для хранения данных определена коллекция Phones, которая инициализируется в конструкторе.

Кроме этой коллекции в классе определены три команды. Команда MoveToTopCommand поднимает элемент на одну позицию вверх в списку (в реальности это выглядит как перемещение ближе к началу коллекции Phones). Команда MoveToBottomCommand, наоборот, опускает элемент на одну позицию вниз. А команда RemoveCommand удаляет элемент из списка.

В коде C# у главной страницы MainPage определим контекст данных:

using System;

using Xamarin.Forms;

namespace MvvmApplication

{

public partial class MainPage : ContentPage

{

public MainPage()

{

InitializeComponent();

this.BindingContext = new PhonesListViewModel();

}

}

}

В коде XAML определим список для вывода объектов, к каждому из которых прикрепим контекстное меню:

<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="MvvmApplication.MainPage">

<StackLayout>

<ListView ItemsSource="{Binding Phones}" HasUnevenRows="True">

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<ViewCell.ContextActions>

<MenuItem Text="Вверх"

Command="{Binding Path=ListViewModel.MoveToTopCommand}" CommandParameter="{Binding}" />

<MenuItem Text="Вниз"

Command="{Binding Path=ListViewModel.MoveToBottomCommand}" CommandParameter="{Binding}" />

<MenuItem Text="Удалить"

Command="{Binding Path=ListViewModel.RemoveCommand}" CommandParameter="{Binding}" />

</ViewCell.ContextActions>

<ViewCell.View>

<StackLayout>

<Label Text="{Binding Title}" FontSize="Medium" />

<Label Text="{Binding Company}" FontSize="Small" />

<Label Text="{Binding Price}" FontSize="Small" />

</StackLayout>

</ViewCell.View>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

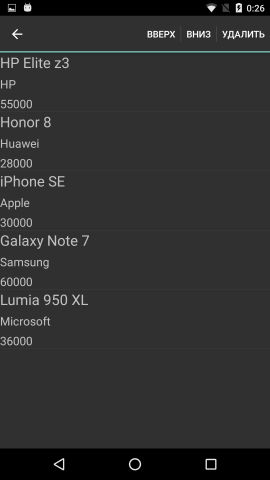
</ListView>

</StackLayout>

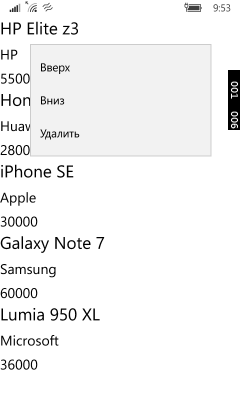
</ContentPage>

Для определения меню применяется свойство ContextActions объекта ViewCell, который представляет собой отдельный элемент. Каждый пункт в этом меню представляет элемент MenuItem. У MenuItem можно определить команду и параметр, который будет пределаться команде при ее выполнении. В данном примере будут вызываться команды из PhonesListViewModel, которым будет передаваться текущий объект, для которого вызвано контекстное меню.

И если мы запустим приложение и одним долгим касанием нажмем на какой-нибудь элемент списка, то отобразится контекстное меню:



При этом надо отметить, что на разных ОС (даже на разных вериях одной ОС Android) визуально контекстное меню может выглядеть иначе. Например, на Windows 10 Mobile:



Хотя здесь элементы меню используют команды и их параметры, однако нам в принцпе не обязательно использовать контекстное меню именно в контексте MVVM. Так, у каждого элемента меню есть стандартное событие Clicked, для которого мы можем определить в коде обработчик. Например:

<MenuItem Text="Show" Clicked="Show" />

А в коде прописать какой-нибудь метод обработки:

private async void Show(object sender, EventArgs e)

{

await DisplayAlert("Контекстное меню", "Пункт Show", "OK");

}

В тоже время в MVVM использование контекстного меню имеет свои плюсы, в частности, возможность передачи параметра.