Table of content:

Introduction to Xamarin.Forms 4
Installation and Create your first app 4
Life Cycle 7
U.I Controls of Xamarin.Forms 7
Pages 9
Layout 11
Views 15
Cell 23
Images, Icons and Splash Screen 24
Pop-Ups in Xamarin.Forms 37
Data Binding in Xamarin.Forms 43
Navigation 46
List View 51
Functionality of a list view: 58
Table View 66

Preface:

This book is written for beginners and covers the basic topics required to learn Xamarin from scratch. The book is not much theoretical but uses practical approach and you can see many code snippets and output images in each section. By reading this book, you will learn to create Xamarin.Forms app and understand the basic of Xamarin.Forms.

About Author:

UMAIR HASSAN is the author of this book. He is an MVP at C-sharp Corner. You can visit his c-sharp corner profile and read his other articles or contact him if you need any help or give any suggestions.

Email: umair.hassan03@gmail.com

Introduction to Xamarin.Forms

Xamarin.Forms is used to make cross-platform mobile applications. You can make mobile apps for Android, iOS, and Windows by writing a shared code. In this book, we are focusing on Xamarin.Forms portable class library (PCL). The real beauty of Xamarin.Forms comes when you learn that you can write a single code only and all three applications are developed by that code. That means your user interface and business logic is shared in Xamarin.Forms. In Xamarin.Forms, most of your code is shared in all applications, however, some code that includes database connectivity is done in each application project and is not shared. Xamarin apps are fully native so you can enjoy the native performance of applications with shared code.

Technology:

Xamarin.Forms uses XAML for user interface and C# to handle its business logic. You can use C# also to make the UI but here, we are using XAML only to create the UI.

Tools:

You can use Xamarin Studio or Visual Studio for development. In the examples below, we are going to use Visual Studio 2017 on Windows 10 OS with Xamarin workload installed in it.

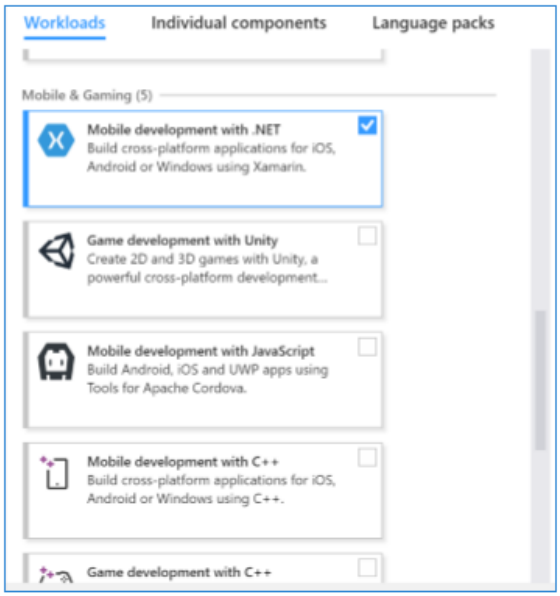
Benefits of Xamarin.Forms:

- Native apps
- Shared Business Logic
- Shared UI
- One Xamarin development team can develop apps for multiple platform
- Less development time

Installation and creation of the first app

Download Visual Studio Installer from here.

First, you need to install Visual Studio 2017 with Xamarin workload.

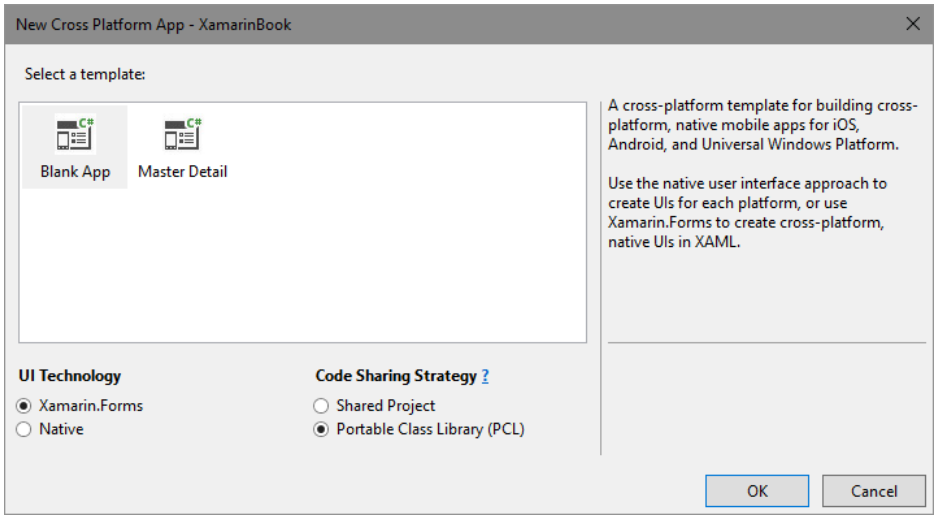


„

Make sure that “Mobile development with .NET” option is checked. By installing it, all the required components are installed in your Visual Studio.

Let’s create our first cross-platform application using Visual Studio 2017. You can follow these simple steps to develop your first app.

1- Open Visual Studio.
2- Go to File -> New Project
3- Go to Cross Platform section -> Cross Platform App (Xamarin) -> rename it -> click OK.
4- Template: Blank App, UI Technology: Xamarin.Forms, Code Sharing Strategy: Portable Class Library (PCL).



5- Select targeted and minimum version for Universal Windows Project.

So, our first project is created. You can see 4 projects in the Solution Explorer - Portable project, Android, iOS, and Universal Windows project. Now, we are ready to start development.

Life Cycle

Every application has its life cycle; it can start, end, and perform different tasks in its life cycle. The life cycle of Xamarin.Forms application also depends on some methods. Expand the portable project, then expand file named “App.xaml”. After this, you can see “App.xaml.cs”. This file contains methods for application lifecycle. Your Xamarin.Forms application revolves around these methods.

```
protected override void OnStart()
{
    // Handle when your app starts
}

protected override void OnSleep()
{
    // Handle when your app sleeps
}

protected override void OnResume()
{
    // Handle when your app resumes
}
```

The OnStart() method is called when your application starts. Whatever you write in this method will be executed when the application is starting.

OnSleep() method is called when you hide your application or minimize it in a way that it is running in the background.

OnResume() method is called when your application comes to the normal state after sleep. When you came back to the application, this method is called.

UI Controls of Xamarin.Forms

We use XAML to make user interface for Xamarin.Forms application but you should keep in mind that Xamarin XAML is different from Microsoft XAML. So here, we are discussing the code of UI elements of Xamarin.Forms.

There are four main control groups of Xamarin.Foms UI, as given below.

1. Pages
   Pages refers to the main page, i.e., which type of page you are using. For e.g. - Content page, Navigation page.

2. Layout
   Layout is the layout you are using in your page; e.g. Stack Layout, Grid Layout.

3. Views
   Views are the single UI elements which you are using in your layout; e.g. button, label.

4. Cell
   Cells are used to present the data in the Table View and List View; e.g. ImageCell, TextCell.

Pages

- *Content Page*
  It can display a single View. You can place any layout or View in it. You can use content page to show any View with a single element. This element can be anything like label, an entry, or you can also use layout in the content page. It doesn't matter that your layout contains more than one element; it can be placed in a content page. Layouts are also discussed below.

- *Navigation Page*
  Use it to make navigation between the pages. i.e., move from one page to another. Most of the mobile applications have more than one page. Thus, you must use navigation page in your application to move from one page to another.

- *Tabbed Page*
  Navigate from one page to another via tabs. This is also a type of navigation page. In this page, you have multiple tabs, all the tabs are in front of you, and you can move from one tab to another easily. For e.g., Instagram is based on Tabbed page.

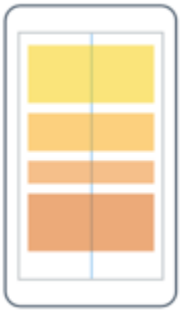
- *Master Detail Page*
  A page that manages two panes of information. Master Detail page contains two panes. Let's discuss an example of it that you have a list of contacts. When you click on a specific contact to see its details, all the details of it are shown on the same page on second pane.

- *Template Page*
  It can display full screen content with a control template and the base class for Content page.

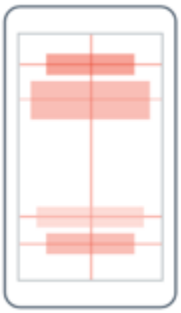
- *Carousal Page*
  Navigate from one page to another via swipe gesture. This is another type of navigation page but in Carousel page, you can swipe to see the next page. Swipe left or swipe right to navigate between the pages, as shown below.

Layout

- *Stack Layout*
  It can place elements in stack, either in vertical or in horizontal orientation. In vertical orientation, you can place all the elements vertically. If you set the orientation to horizontal, then all its elements are automatically placed in horizontal way. This is the most simple and easy to use layout. You can also see its example here.



- *Absolute Layout*
  This layout is used to position its child elements at absolute requested positions. Absolute layouts are used when you want to move away from all the layouts restrictions and make a layout totally different. We can use absolute and proportional values to set height, width, X-Axis and Y-Axis of an element. You can see its example here.



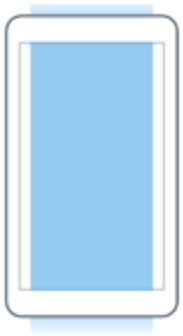
- *Grid Layout*
  Divide layout in rows and columns. Your full screen is divided into rows and columns and you can utilize them to make your desired View. Most Grid layouts are seen in pictures gallery and other Grid layouts. If you want to learn more about Grid layout, visit here.

- *Relative Layout*
  Used to position its elements relative to other element. For example, you have a navigation bar at the top of your application and you want this to be always viewed 20 units down from the main container. Now, you may set its y-axis to 20 units down relative to its container. To see practical implementation, click here.

- *Content View*
  An element with a single content. ContentView has very little use of its own.

- *Scroll View*
  It is used for scrolling a page. If you want to show more and more data on an Application page, then don't take any tension; Scroll View is always there to help you. You can scroll

screen up and down to view more data, as show in the image given below. E.g. Facebook and other big social media applications require scrolling.



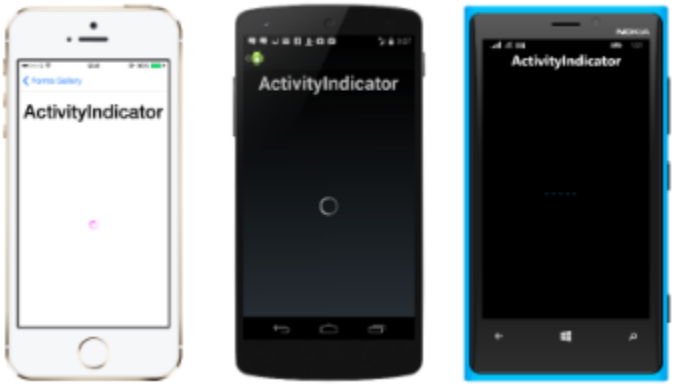
- *Content Presenter*
  A layout manager for templated views. It is used within a ControlTemplate to mark, where the content to be presented appears.

- *Frame*
  It contains a single child with framing options. Frame has a default Layout and Padding is of 20.

- *Template View*
  An element, which displays content with a control template and the base class for ContentView.

Views

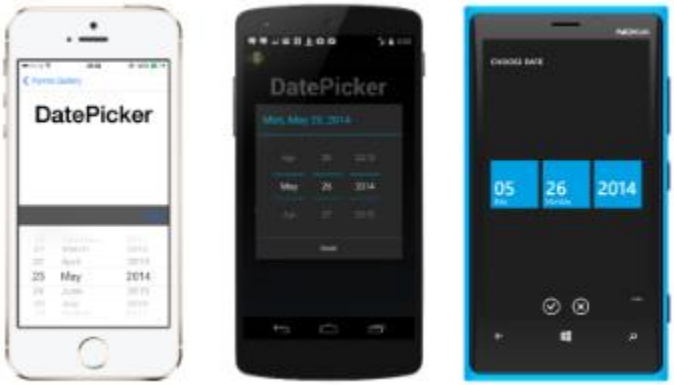
- *Activity Indicator*
  It is used to indicate that something is working. You can see small circle rotating in an image shown below. This is the sign of an Activity Indicator. It is used to satisfy the user that our Application starts working on its given command.



- *BOX View*
  A simple box with solid colors. Change box size, color and other properties to make it more attractive.

- *Button*
  A simple button that can perform any action, which you set on the clicked event of a button. On button, you can set navigations and different task to be done according to your Application logic.

- *Date Picker*
  It shows a calendar, which is used to pick a date. Date Pickers are mostly seen on Calendar apps, where you can set the events on Date Picker or pick the current date that the user sets it on Picker.



- *Editor*
  It is a multiple line Text Editor. If you are familiar with an HTML, then you can say an Editor is a text area, which is used to get multiple line inputs from the user. Editor is used to edit multiple lines e.g. when you want long address or the values from the user , than an Editor is used.

- *Entry*
  Edit single line to text. It is also used to take an input from the user. It is mostly used when small input is required from the user. E.g. if you want a user E-mail or a password,

then an entry is the best suitable option for you. In the picture given below, you can see the entry element.

- *Image*
  It is use to show any image. You can make your application look and fell better by using attractive images. Or you may use images to showcase you products in your Application or many more.

- *Label*
  It can view some text. Any type of text or description is displayed in a label. Labels makes the users to understand your Application more easily.

- *List View*
  It is used to present the list of data. You can use different data templates in the list. i.e. TextCell or ImageCell. TextCell contains text and its description whereas an ImageCell contains an image and with a TextCell, you can make you own ViewCell to display the data in your list, as per your desire.

- *OpenGL View*
  Works only for iOS and Android projects.

- *Picker*
  A view control is used to pick an element from the list.

- *Progress Bar*
  It is used to show the progress. On each step of your backend work, you can set Progress Bar value. Progress Bar is used to satisfy the user and makes him aware that his work is progressing. It can show the percentage of work done and remaining work to the user.

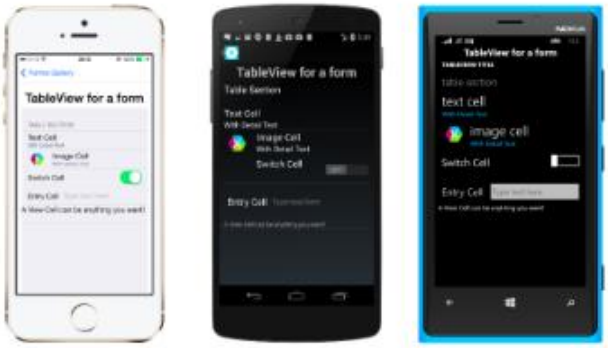
- *Search Bar*
  It is used to search the data from a list or anything. You can place Search bar to make search from a web or from anything. Most of us daily use Search Bar on Google home page.

- *Slider*
  A control that inputs a linear value.

- *Stepper*
  It is control that inputs a discrete value, which is constrained to a range.

- *Switch*
  Toggle button/ on or off button. It is used when you give the user only two choices 'Yes' or 'No'. In real word, we can also use switches to on or off our appliances. This switch is also something like that.

- *Table View*
  It contains rows and columns. It contains rows and columns to place your data in more presentable way.



- *Time Picker*
  Time Picker is used to pick a time, which is mostly seen in an Alarm Clock Application and other Application, where we want to set an event or schedule anything.

- *Web View*
  It is used to present Web data or HTML content.

Cell

Cells are the specialized elements, which are used to present a date in List view and table view. Cells are the elements, which are specially designed to be added in ListView and TableView Controls.

Different types of cells are shown below.

- *Entry Cell*
  A cell that contains a label and single line entry field.

- *Image Cell*
  A cell that includes an image in it.

- *Text Cell*
  A cell is the one, which shows the text and the detail.

- *Switch Cell*

It is a cell with a label and on/off switch.

Images, Icons and Splash Screen

There are two types of images, which are given below.

- Platform Independent (Backgrounds, Images)
    - Download using an URL.
    - Embed in a Portable Class Library (PCL).
- Platform Specific (Icons, Splash Screens)
    - Add to each application Project

First Section - Platform Independent Images

Let's start with platform independent images.

In our Xamarin.Forms project, we use platform independent images by downloading them from URL or by embed image in our project.

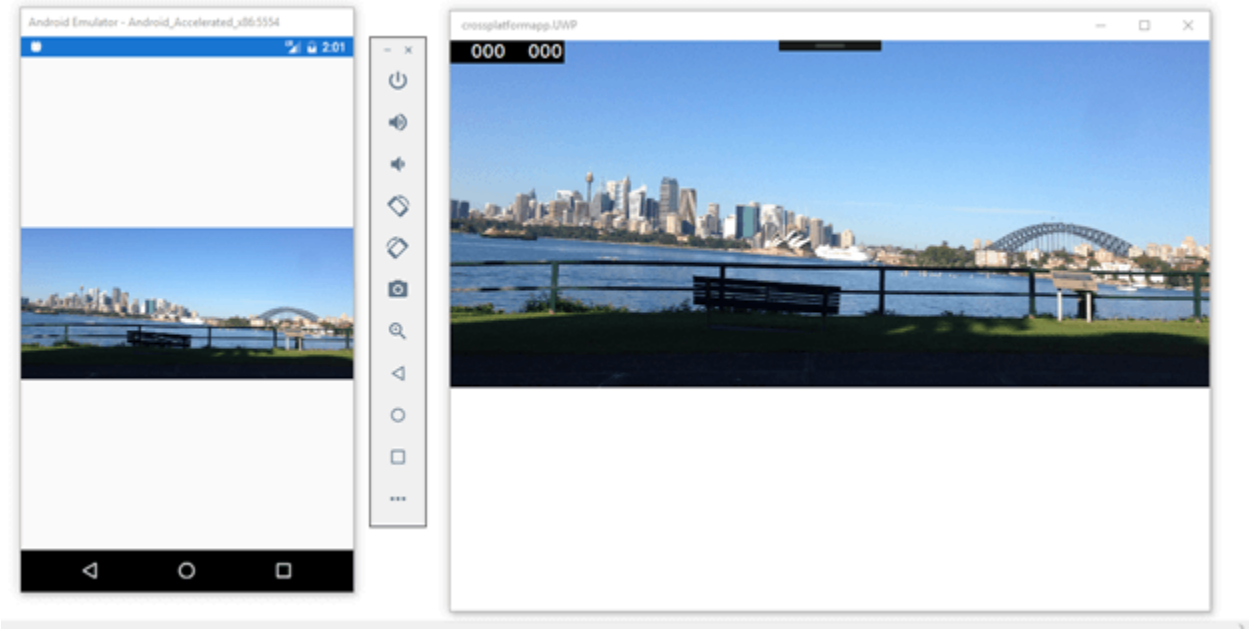
Download using a URI

We can simply download an image from URL by adding URL in image source. You can add image from XAML or from code. Both examples are given.

Via XAML

1. <Image x:Name="tempimage" Source="https://stepupandlive.files.wordpress.com/2015/03/sydney.jpg"></Image>

Output on Android and Windows.



Via Code Behind

Now, add the same image via code file. We can add image URl from code behind like this.

XAML

1. <Image x:Name="tempimage"></Image>

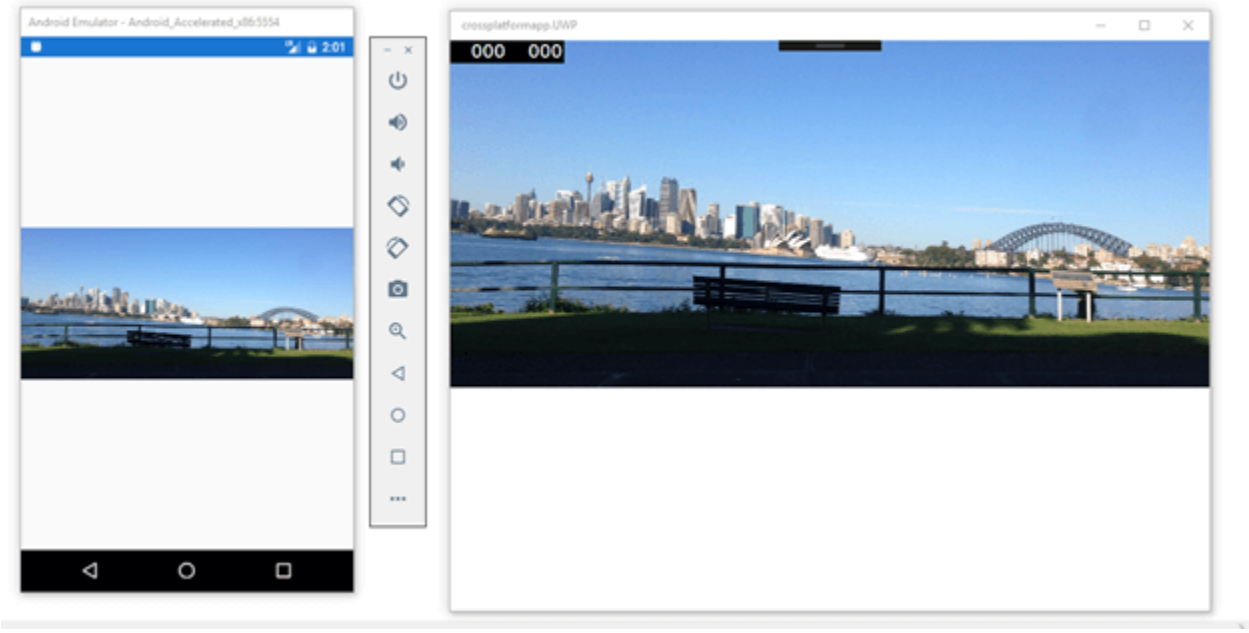
Image source is added from code file.

Code File

1. **var** imagesource = **new** UriImageSource { Uri = **new** Uri("https://stepupandlive.files.wordpress.com/2015/03/sydney.jpg") };
2. tempimage.Source = imagesource;

It can directly download image from given source. The same output is generated but this time from code file.

Output on Android and Windows

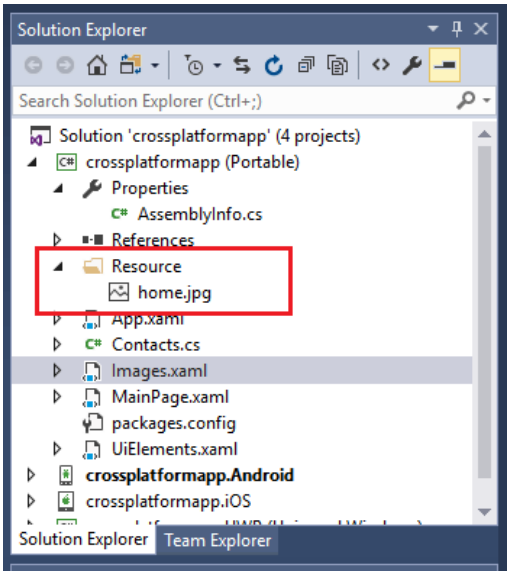


Embed in a Portable Class Library (PCL)

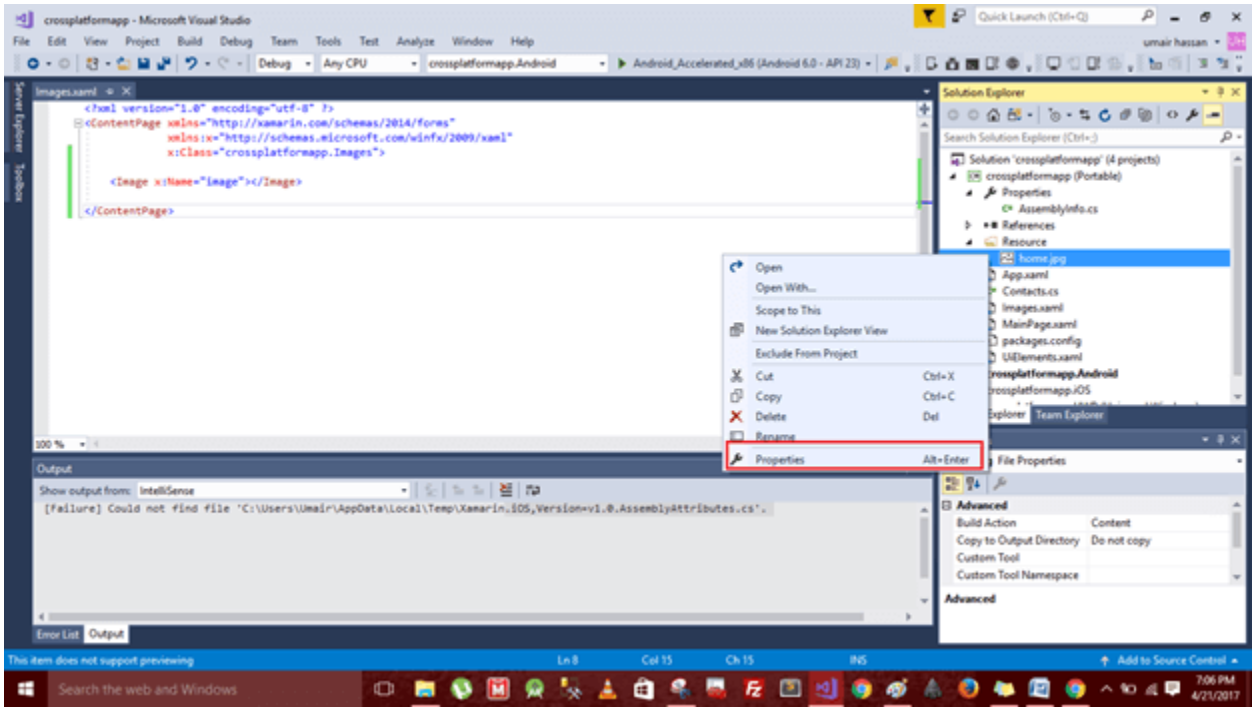
Let's discuss how to use embedded image in our project.

In Visual Studio

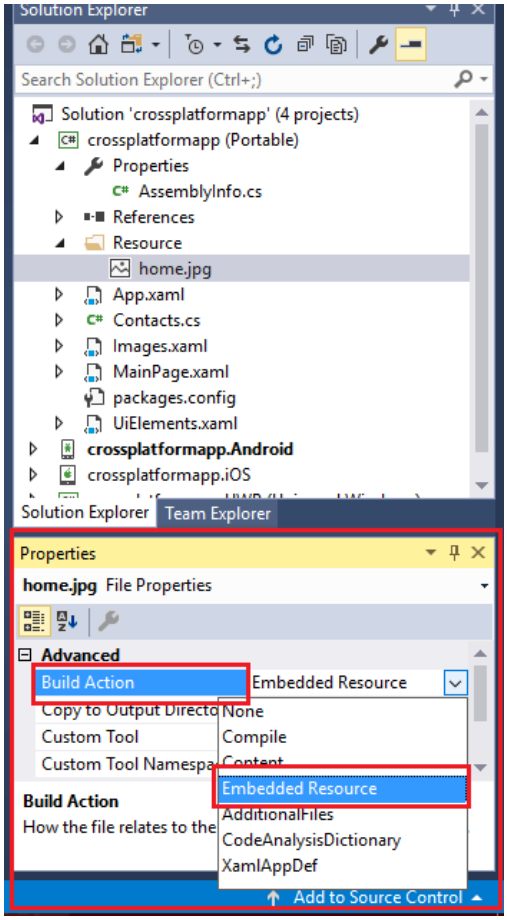
Make a folder of your resources and paste your desired image in it.



After pasting an image right click on image and go to properties.



Look for "Advanced" section and change "Build Action" to Embedded Resource.



After changing its Build Action type, you can use this image in both XAML and from code file.

But to use it from XAML, you have to make a custom markup for it. We will discuss it later. Firstly, this is the code file sample to use embedded image.

Use embedded image from code file

Xaml

1. <Image x:Name="image"></Image>

Code

1. image.Source = ImageSource.FromResource("crossplatformapp.Resource.home.jpg");

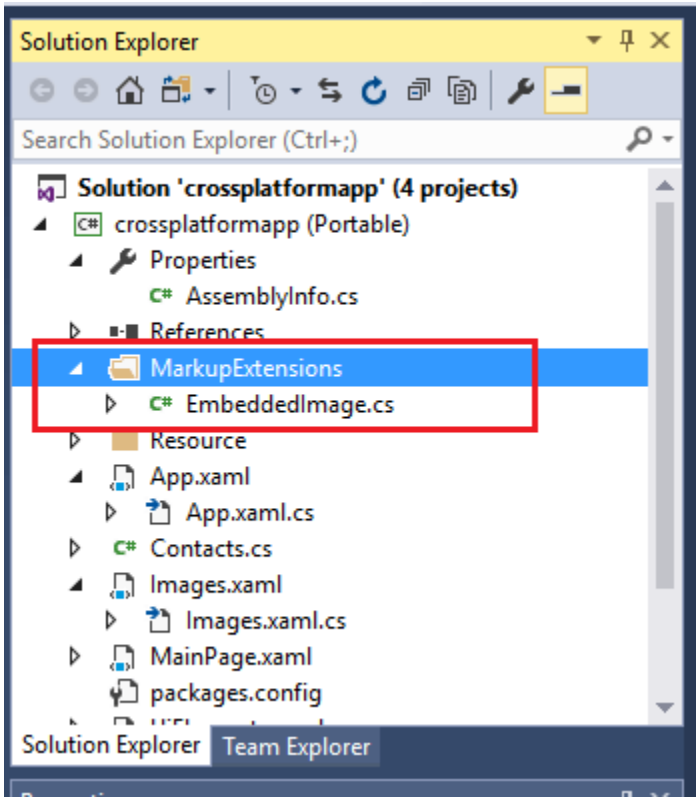
Add complete path of file in this format:
(Projectname.foldername.imagename.ImageExtension)

Output on both android and windows

Include Embedded Image Source in XAML

Now, use the same image from XAML. You can't directly put image source in the format given above. Firstly, you have to make custom markup class for this.

Make new folder of MarkupExtensions and add new class named EmbeddedImage.cs.



Now, implement interface "IMarkupExtension" to this class.

```
1.  using System;
2.  using Xamarin.Forms;
3.  using Xamarin.Forms.Xaml;
4.  namespace ProjectName.MarkupExtensions {
5.     public class EmbeddedImage: IMarkupExtension {
6.        public string ResourceId {
7.           get;
8.           set;
9.        }
10.       public object ProvideValue(IServiceProvider serviceProvider) {
11.          if (String.IsNullOrWhiteSpace(ResourceId)) return null;
12.          return ImageSource.FromResource(ResourceId);
13.       }
14.    }
15. }
```

Here, our custom markup class is complete.

Now, use this in XAML.

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:ProjectName.MarkupExtensions" x:Class="Practice__app.ImagePage">
3.     <StackLayout>
4.         <Image Source="{local:EmbeddedImage ResourceId=crossplatformapp.Resource.home.jpg }"> </Image>
5.     </StackLayout>
6. </ContentPage>
```

Firstly, add xmlns namespace just like this.

xmlns:local="clr-namespace:ProjectName.MarkupExtensions"

Change ProjectName to your application project name

Then, give image source like shown above to use the image.

Same output is shown but this time we use XAML to use embedded image.

Output on both Android and Windows

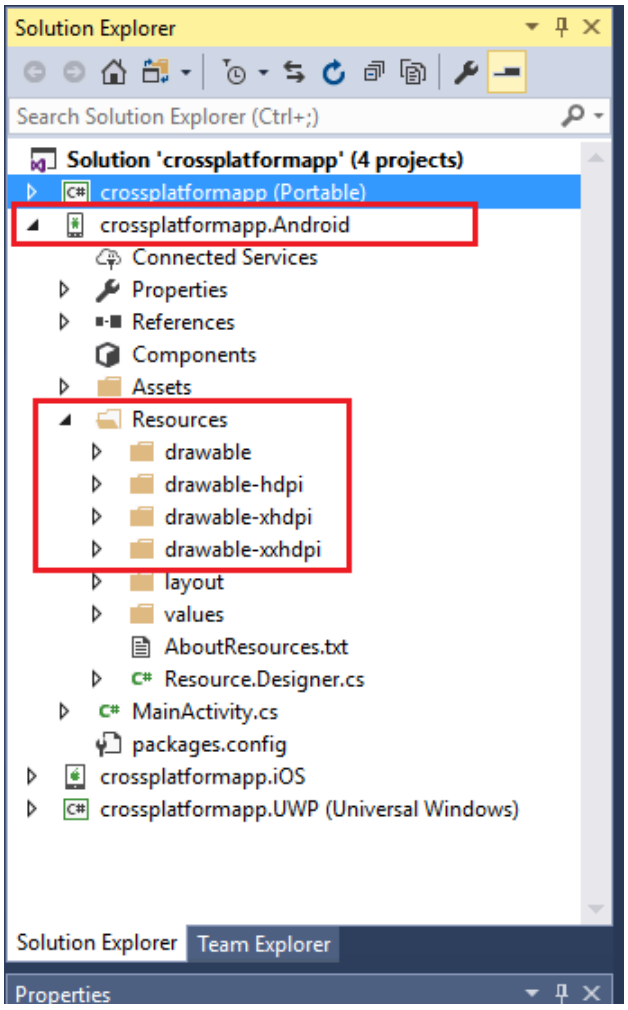
Here, our first section is complete -- now start with platform specific images. We can make a native look of the application by putting images in folders of each application project.

You can add image in Android, Windows, and iOS project to use platform specific images.

For ANDROID

Explore Android folder in your Xamarin.Forms project, then expand resources folder. Here, you will see the following folders.

- drawable
- drawable-hdpi
- drawable-xhdpi
- drawable-xxhdpi



Here, hdpi stands for high dpi; xhdpi stands for extra high dpi; xxhdpi stands for extra extra high dpi.

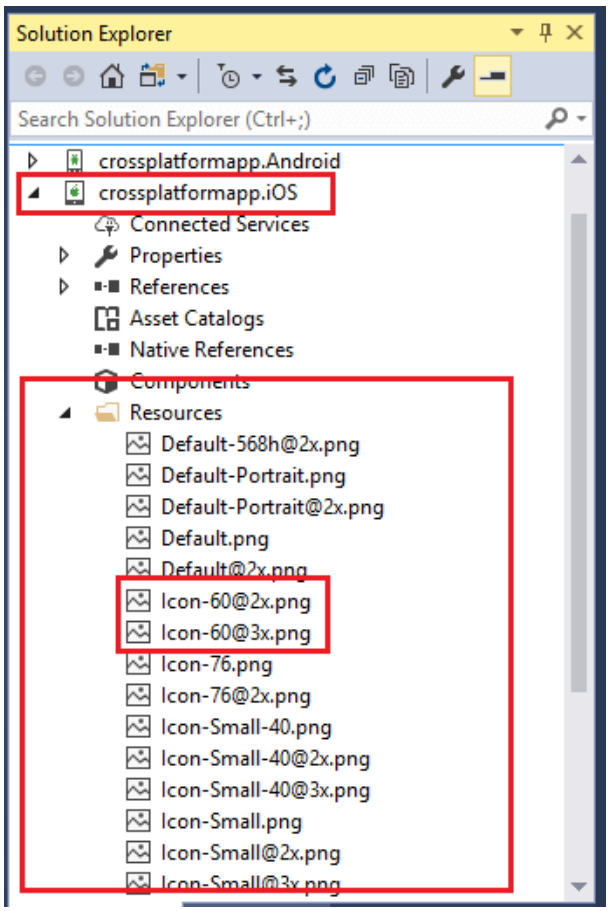
You can put your original file in drawable folder, and if you have high dpi version of this image, put this in drawable-hdpi. Similarly, for extra high dpi version of this image, you may put this image in drawable-xhdpi folder. Keep in your mind that image file name in each folder remains the same.

FOR iOS

For iOS project, expand the Resources folder and paste your images here. Here, a question arises - How iOS differentiates high quality version of image?

In iOS, we follow a naming convention to differentiate our images.

If your image name is "abc.png", for high version of this image, you may use "abc@2x.png" or "abc@3x.png".



Keep in mind that in iOS, all images are in same folder but with different naming convention

FOR WINDOWS

Expand Windows folder in your Xamarin.Forms project and paste the image there. Don't paste image in any folder but just in the project file.

Now, we add images in all platforms. Let's use them in our portable Xamarin.Forms project and give native look and feel to our project.

Suppose you paste an image name "abc.png" in all three project folders, to use this in XAML file, we can code like this:

1. <Image Source="abc.png"></Image>

Now you can directly use image source here without using any external markup class or any embedded image.

Application Icons and Splash Screens

For Visual Studio

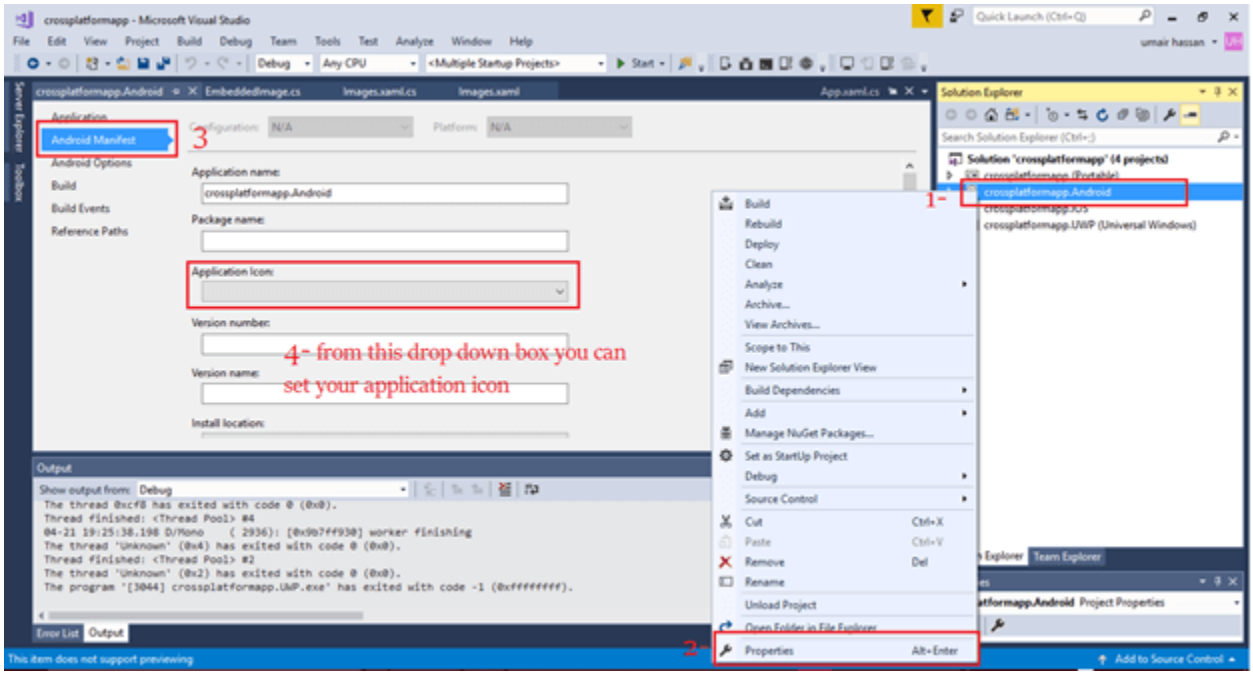
Android

Right click on your Android project.

Go to Properties and search for Android Manifest. Here, you see Application Icon drop down box.

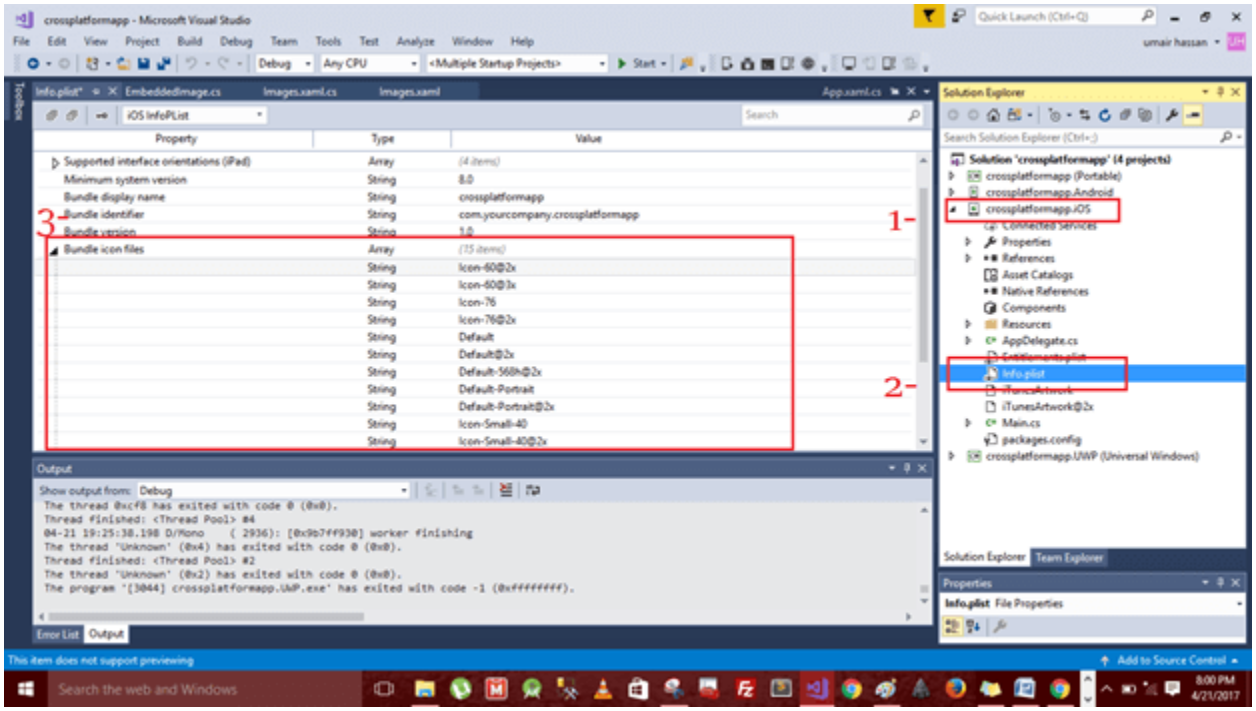
Open this dropdrown.

All images and icons in your drawable folder are shown. You can use any of this as your application icon.



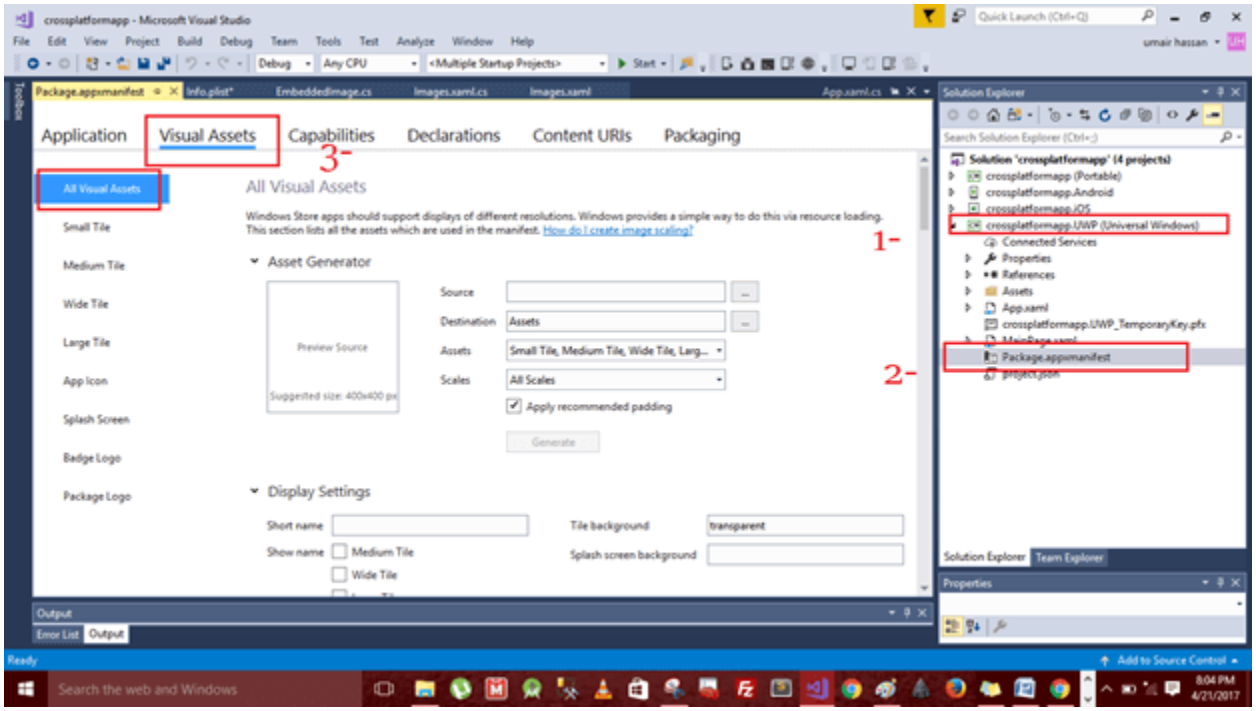
iOS

Expand iOS folder and double click info.plist file. Here, you see Bundle icon files; default icons are set there. You can change them according to your desire.



Windows

For Windows project, open Package.appxmanifest and open Visual assets. Set all visual assets of your Windows application.



This is how you can set icons and splash screen in Android, iOS and Windows project of your Xamarin.Forms portable project.

Pop-Ups in Xamarin.Forms

Here, we are going to discuss two types of Pop-ups - Display Alert and Action Sheet.

Display Alert

To display alert in Xamarin.Forms, you have to use DisplayAlert Method on any page. Here, we are going to use a single button in XAML and on clicked event of button, we can simply show alert.

The required code and output are given below.

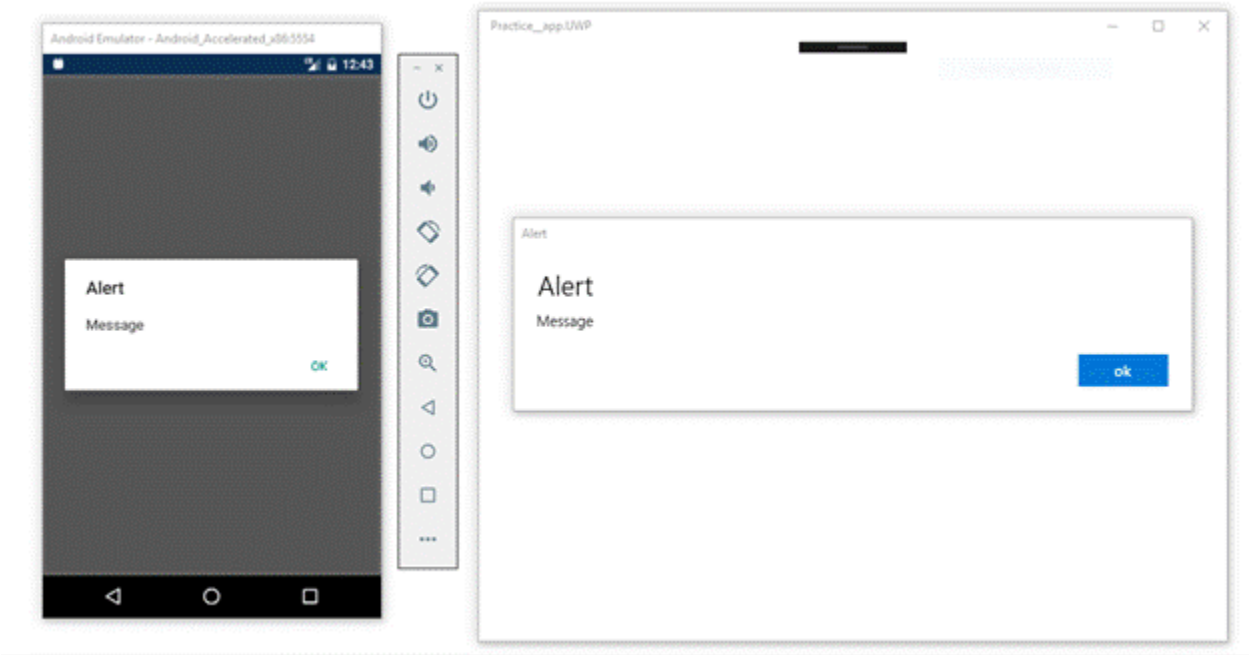
XAML

```
<Button xName="alert" Text="Alert" Clicked="alert_Clicked"></Button>
```

Code

```
private void alert_Clicked(object sender, EventArgs e)
{
    DisplayAlert("Alert", "Message", "ok");

}
```

Output on Android and Windows desktop



You can also use two buttons on alert. One is for accepting and the other one is Cancel. Just add the Cancel button in previous code. Keep in mind that all values in Display alert method are always in string.

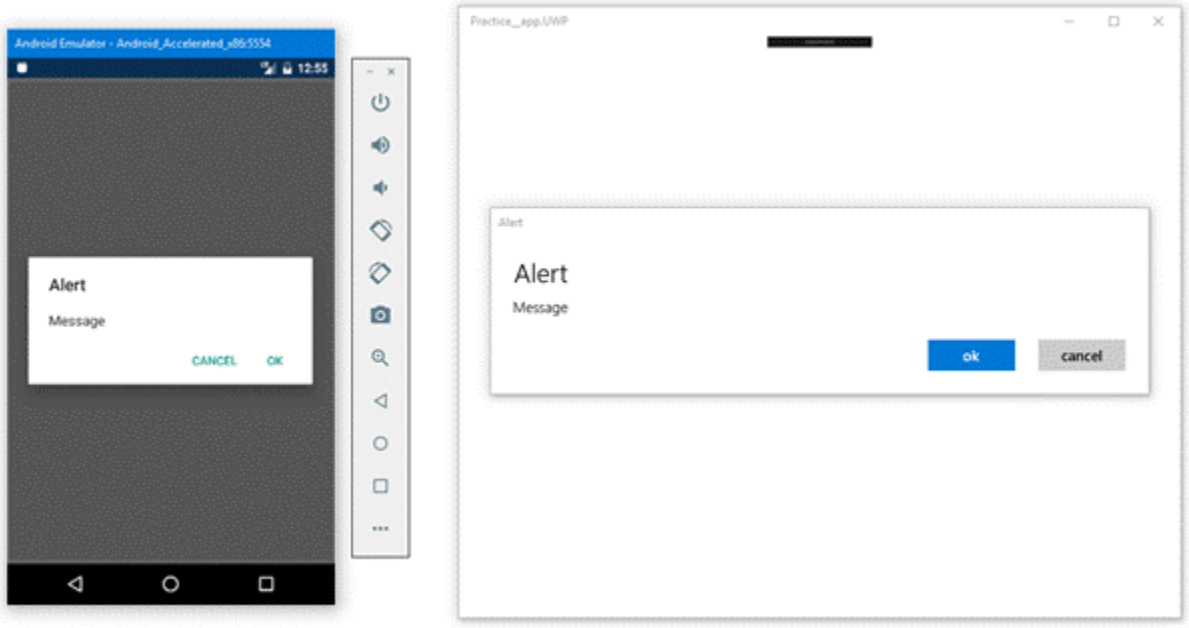
XAML

```
<Button xName="alert" Text="Alert" Clicked="alert_Clicked"></Button>
```

Code

```
private void alert_Clicked(object sender, EventArgs e)
{
    DisplayAlert("Alert", "Message", "ok","cancel");

}
```

Output on Android and Windows desktop



And if you want to save user response or utilize user response that is given by alert method, then you can do this by using await. Result given by display alert method is a Boolean value. When user clicks on first button response, the result is true and when user selects second button, it gives false. Let's move towards a practical example.

XAML

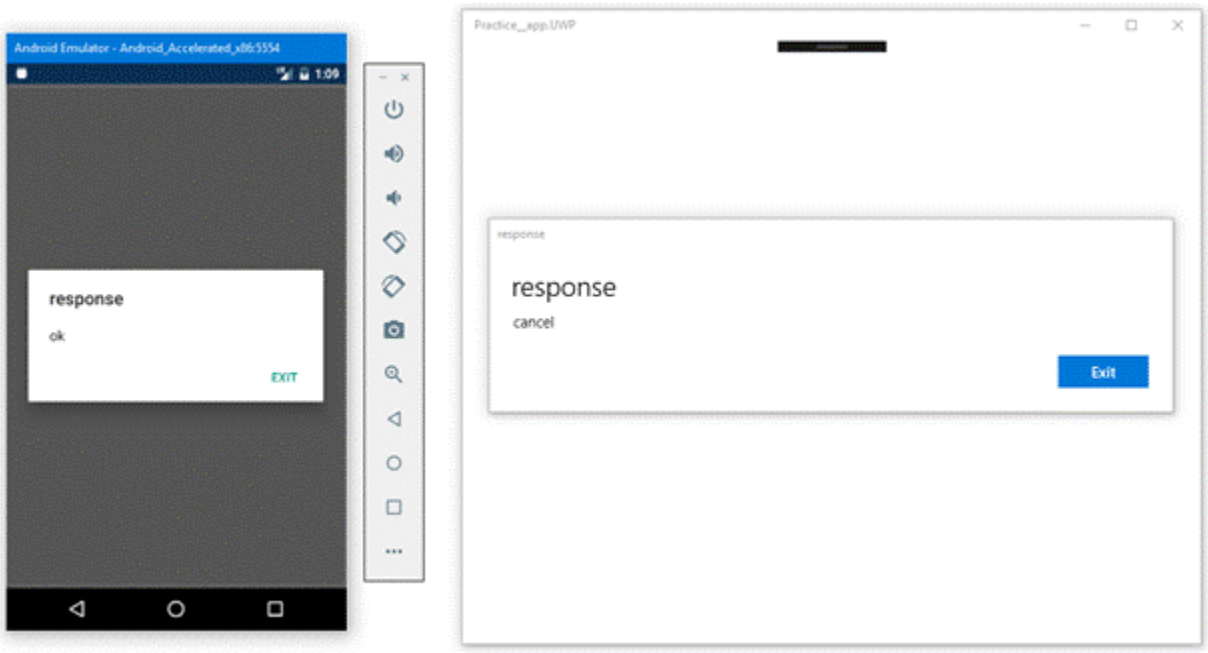
```
1. <Button xName="alert" Text="Alert" Clicked="alert_Clicked"></Button>
```

Code

```
1.  private async void alert_Clicked(object sender, EventArgs e)
2.      {
3.       var response = await DisplayAlert("Alert", "Message", "ok","cancel");
4.
5.        if (response)
6.        {
7.           //user click ok
8.          await DisplayAlert("response","ok","Exit");
9.        }
10.
11.       else
12.       {
```

```
13.             //user click cancel
14.             await DisplayAlert("response", "cancel", "Exit");
15.
16.         }
17.     }
```

Output on Android and Windows desktop



Both of the conditions are shown in output image. In Android device, user selects OK, so response is generated from if condition. And in Windows desktop application, user selects Cancel, so the response is generated from else section.

Here, we are done with the display alert. Now, let's move toward Action Sheet.

Action Sheet

Action sheet is used when you give user multiple options to select from.

Action Sheet is also a method that contains string values. First value is title, second value is cancel, third one is destruction, and after this, you can put multiple actions.

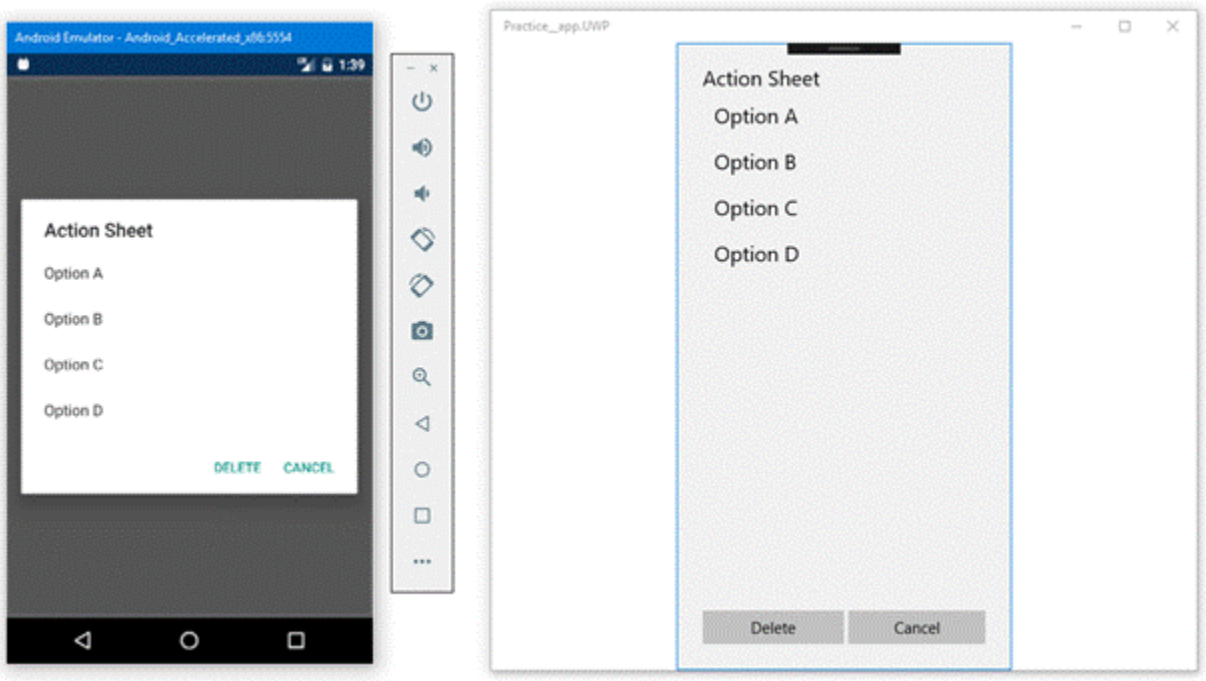
XAML

```
1. <Button xName="alert" Text="Action Sheet" Clicked="alert_Clicked"></Button>
```

Code

```
1. private void alert_Clicked(object sender, EventArgs e)
2. {
3.     DisplayActionSheet("Action Sheet","Cacnel","Delete","Option A", "Option B","Option C", "Option D");
4. }
```

Output on Android and Windows desktop



You can also utilize action sheet response and do some work on it. Response generated by action sheet is a string value. You may utilize it by if conditions. But for now, I just simply show a display alert that contains its response. The code is given below.

XAML

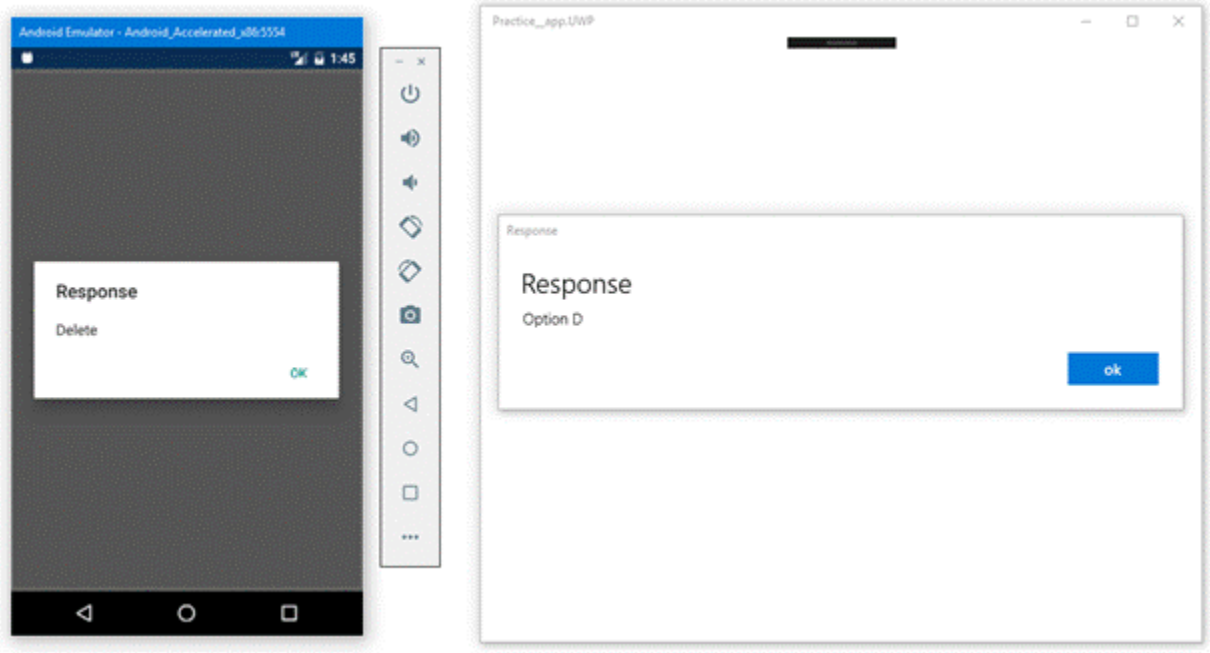
```
1. <Button xName="alert" Text="Action Sheet" Clicked="alert_Clicked"></Button>
```

Code

```
1. private async void alert_Clicked(object sender, EventArgs e)
```

```
2.  {
3.   var response = await DisplayActionSheet("Action Sheet","Cancel","Delete","Option A",
     "Option B","Option C", "Option D");
4.    await DisplayAlert("Response", response,"ok");
5.  }
```

Output on Android and Windows desktop



This is how we can capture user response in action sheet.

Data Binding in Xamarin.Forms

Data binding is the process, which establishes a connection between the Application user interface and Application logic. We may bind the data with the elements and if binding is done correctly, it can reflect changes automatically.

The main benefit of Data binding is that you no longer have to worry about synchronizing the data between your views and data source.

Binding has several properties including.

- Path
- Mode

Binding Path

Path is used to specify property name of the source object, which is used for binding.

Binding Mode

Mode is used to provide the direction in which property value changes are made.

- *OneWay*
  In One way binding, the changes are made from its source to the target.

- *TwoWay*
  In Two way binding, the changes are made in both the directions and Two way binding makes sure that the source and the target are always synchronized.

- *OneWayToSource*
  Changes are made from the *target* to the *source* and are mainly used for read-only bind able properties. In Xamarin.Forms Mode property is default set to OneWay.

Binding example

Here, we can link properties of two views on a single page.

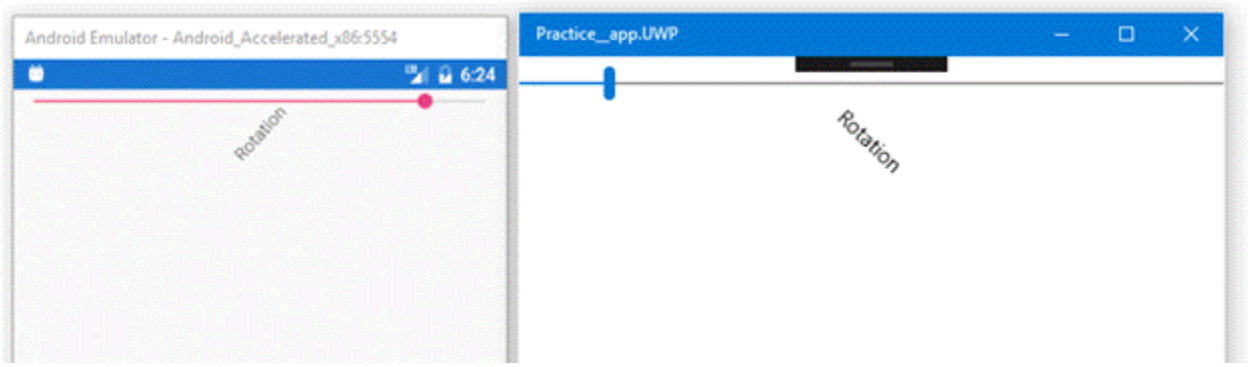
Let's go through its example,

XAML

```
1.  <StackLayout>
2.     <Slider x:Name="slider" Maximum="360"></Slider>
3.     <Label Text="Rotation"
4.           BindingContext="{x:Reference Name=slider}"
5.           Rotation="{Binding Path=Value}"
6.           HorizontalOptions="CenterAndExpand"
7.           ></Label>
8.  </StackLayout>
```

Here, we bind the value property of slider with rotation property of the label with Oneway binding, which is set by default, so, when we move the slider, label starts rotating and maximum value of slider is set to 360.

Output on an Android and Windows desktop is given below.



Navigation

- Introduction to Navigation Page
- Practical example of navigation with output on Android and Windows desktop application
- Make your own Back button on Navigation Page
- How to remove Navigation Bar
- How to disable the device Back button

Introduction

Here we will go through an example of navigation in Xamarin.forms. We will navigate from one page to another and also, will explore its properties to make the navigation better.

Let's start our discussion with defining the Navigation Page.

It is used to make navigation between pages, i.e., move from one page to another. Most of the mobile applications have more than one pages. So, you must use navigation page in your application to move from one page to another.

Navigation page looks like the image below.

Navigation Page

Now, let's go through a simple example and navigate between two pages.

Firstly, we have to make a simple View page. In this example, we will make a View page that contains a button to move towards the next page.

XAML

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
3.              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

```
4.          x:Class="Practice__app.MainPage"
5.          >
6.
7.     <Button x:Name="btn" Text="Next Page" Clicked="btn_Clicked"></Button>
8.
9.  </ContentPage>
```

And on its click-event handler, the code looks like the following.

Code

```
1.  private async void btn_Clicked(object sender, EventArgs e)
2.  {
3.    await  Navigation.PushAsync(new NextPage());
4.  }
```

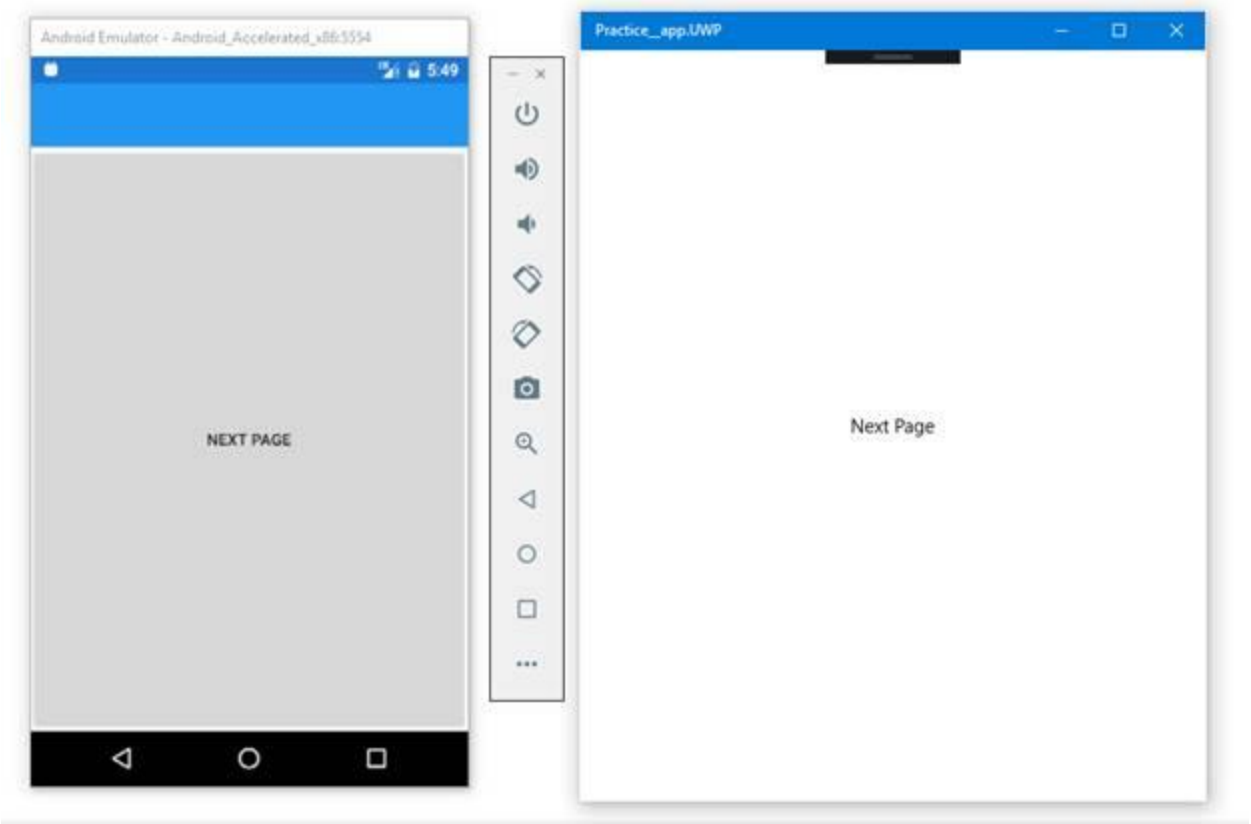
"NextPage" is the name of second page declared in this application. In this application, you have to make two pages - Main page and Second page.

Go to App.xaml.cs and after InitializeComponent(); you have to declare your navigation page like this.

Code

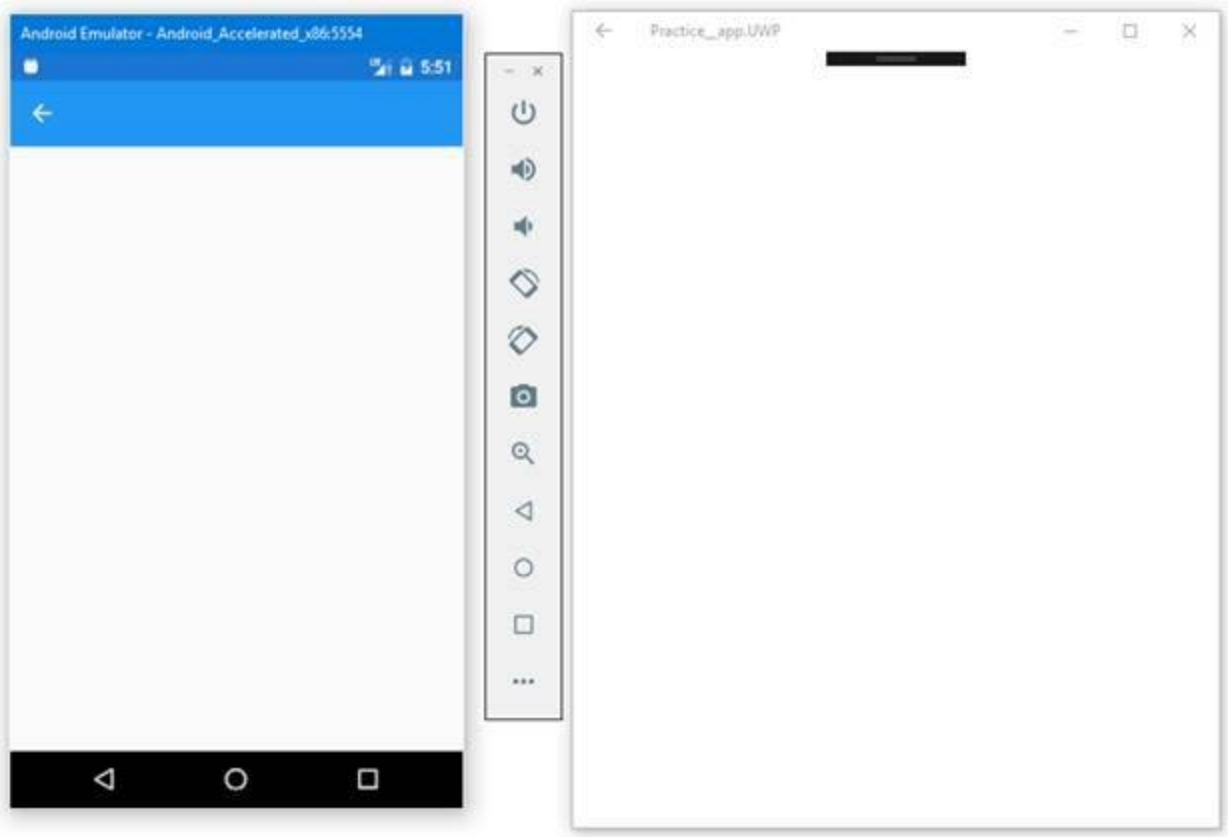
```
1.  MainPage = new NavigationPage(new MainPage());
```

Output on Android and Windows desktop



And by clicking on the "Next Page" button, you are navigated to the next page.

Output when clicked on the Next Page button



Back button

What if you want to set your own Back button on Next Page?

Make a button on next page and set its Click event.

XAML

```
1. <Button x:Name="Back" Clicked="Back_Clicked" Text="Back"></Button>
```

Code

```
1. private async void Back_Clicked(object sender, EventArgs e)
2.     {
3.     await  Navigation.PopAsync();
4.     }
```

For this purpose, we will use PopAsync() method.

Remove Navigation bar

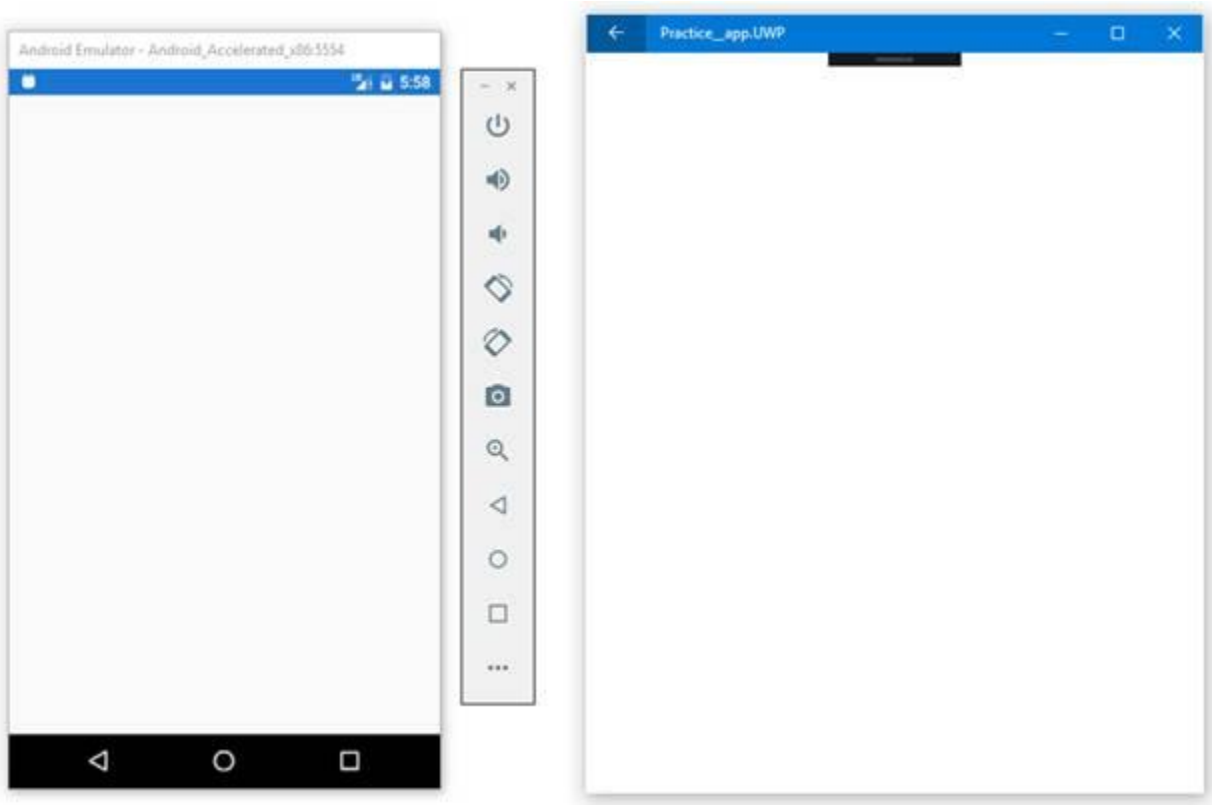
The next page is shown with a bar on the top. That contains Back button. This bar is also shown on your Main page. This is the default behavior of application. But, what if you want to hide that bar?

NavigationPage.HasNavigationBar="False"

This is used to remove the navigation bar from your page. Use this code on the specific page to remove its navigation bar and the XAML looks like below.

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
3.              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4.              x:Class="Practice__app.NextPage"
5.              NavigationPage.HasNavigationBar="False"
6.              >
7. </ContentPage>
```

Output on Android and Windows desktop



Here we go. No navigation bar is displayed on your next page.

Disable Back Button

But wait! There's another problem; user can move to previous page by using back button on Android phone or by using back button displaying on the top of Windows desktop application. If you want the users to not to go on the previous page again, use the following code to disable back button.

For this purpose, you have to override OnBackButtonPressed() method on your NextPage.xaml.cs file.

Code

```
1. protected override bool OnBackButtonPressed()
2.     {
3.         return true;
4.     }
```

After overriding this function now, your user will not move to the previous page by using the back button on the device.

List View

Introduction to ListView

ListView is used to present data in list, especially long lists that require scrolling.

You all are already familiar with list as you are using list of contacts daily in your mobile phone. Other examples include your messaging inbox that contains a list of messages. Let's implement this.

Create a Basic ListView

In our mobile application, we can implement list by gathering data from remote API or by some database. But in this example, we will make a sample hard code list of data and implement it. Now, start creating a basic ListView. The necessary code is given.

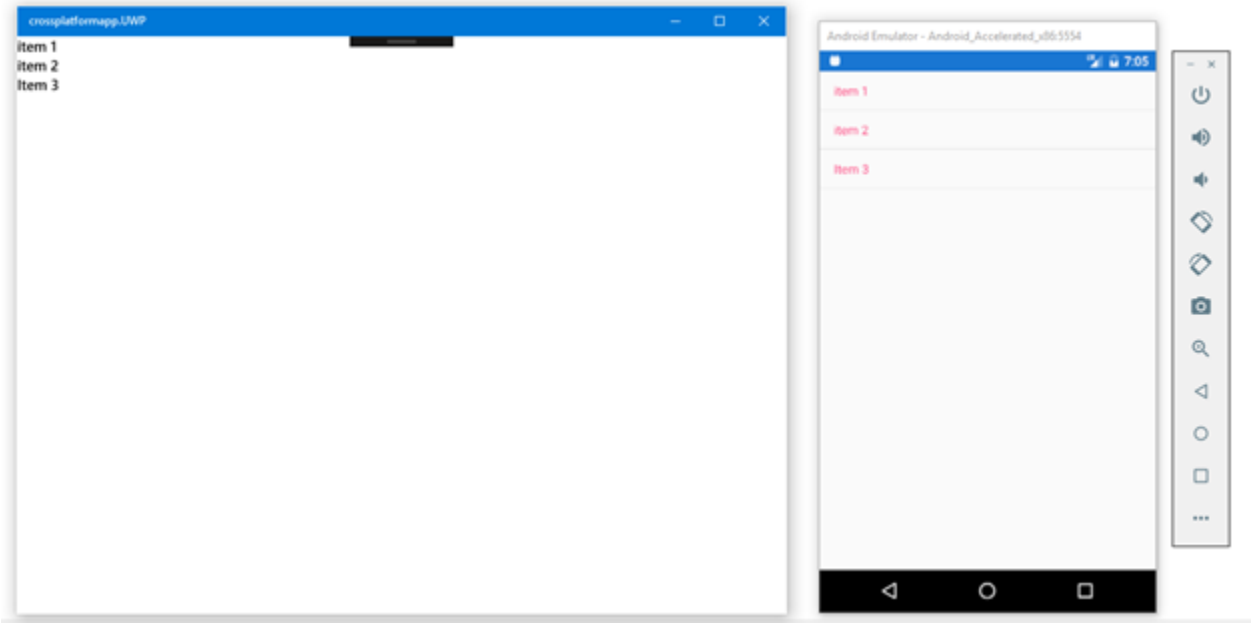
XAML

```
1. <ListView x:Name="lst"></ListView>
```

Code

```
1. lst.ItemsSource = new List<string>() { "item 1", "item 2", "Item 3" };
```

Output on Windows and Android



Binding & Cells

Now, make a complete list and bind its elements. Firstly, we make a class and then bind its properties with ListView.

Add a new class.

```
1.  public class Contacts {
2.      public string Name {
3.          get;
4.          set;
5.      }
6.      public string Num {
7.          get;
8.          set;
9.      }
10.     public string imgsource {
11.         get;
```

```
12.        set;
13.    }
14. }
```

For just practicing purposes, make a class of contacts and define three properties of Name, Num, and imgsource as shown above in the code.

Then, go to code file, add some data, and change item source of ListView.

```
1.  lst.ItemsSource = new List < Contacts > () {
2.      new Contacts() {
3.          Name = "Umair", Num = "0456445450945", imgsource = "http://bit.ly/2oDQpPT",

4.      },
5.      new Contacts() {
6.          Name = "Cat", Num = "034456445905", imgsource = "http://gtty.im/2psFEos",
7.      },
8.      new Contacts() {
9.          Name = "Nature", Num = "56445905", imgsource = "http://gtty.im/2psFEos",
10.     },
11. };
```

As we discussed earlier, there are three types of cells in ListView. Now, let's discuss these three cells and perform some data binding.

TextCell

TextCell is the data template of ListView. In this cell, we can set text and details. Other properties of TextCell include text color, detail color etc.

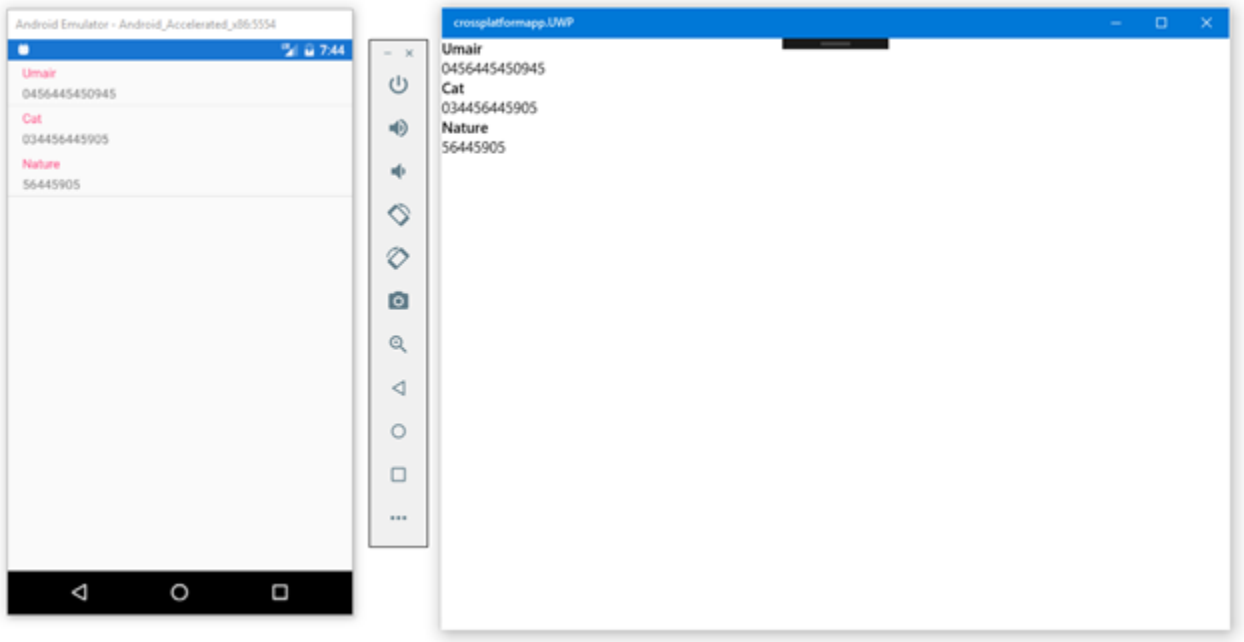
XAML

```
1.  <ListView x:Name="lst">
2.      <ListView.ItemTemplate>
3.          <DataTemplate>
4.              <TextCell Text="{Binding Name}" Detail="{Binding Num}"></TextCell>
```

```
5.        </DataTemplate>
6.     </ListView.ItemTemplate>
7.  </ListView>
```

Here, you can see that we used the TextCell and set binding. We set Binding of 'Text' with 'Name' and we binded the 'detail' with 'Num' property of contact class.

Output on Android and Windows



This is how TextCell is shown on Android and Windows. It will show some text and its details.

ImageCell

Now, let's use an ImageCell. ImageCell is a cell that contains images in it. ImageCell includes ImageSource, Text, and Details.

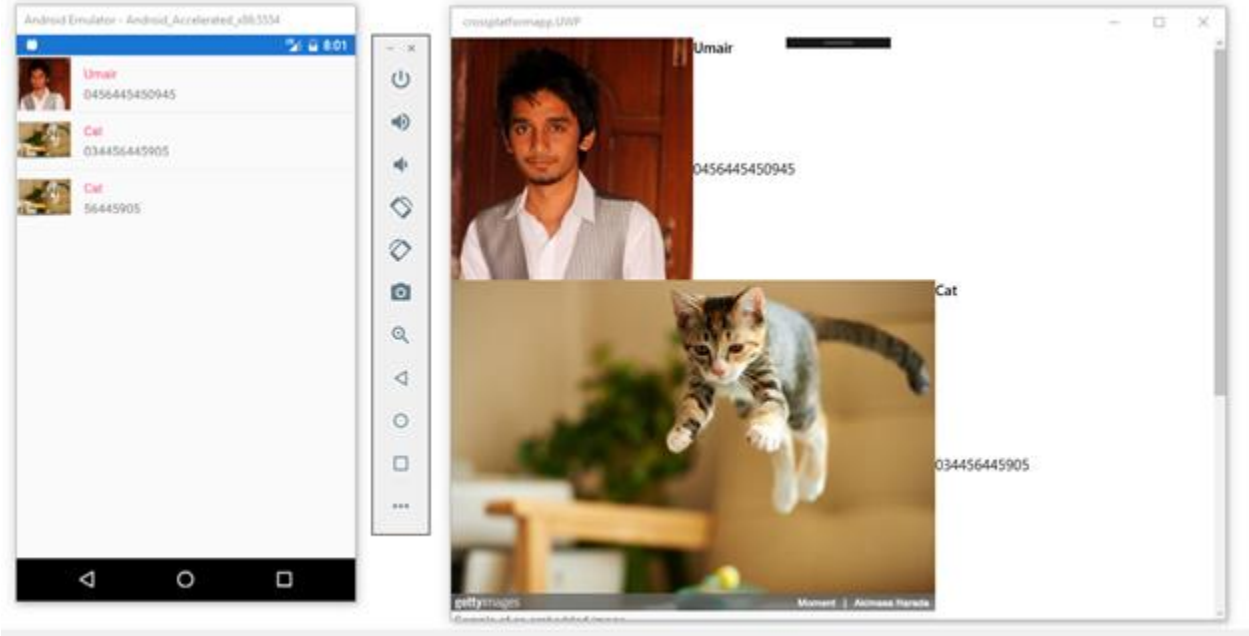
XAML

```
1.  <ListView x:Name="lst">
2.     <ListView.ItemTemplate>
3.        <DataTemplate>
4.           <ImageCell Text="{Binding Name}" Detail="{Binding Num}" ImageSource="{Bindin
     g imgsource}"></ImageCell>
```

```
5.        </DataTemplate>
6.    </ListView.ItemTemplate>
7. </ListView>
```

Here, you can see that we used the ImageCell and set Binding. We set Binding of 'Text' with 'Name', 'detail' with 'Num' and 'ImageSource' with 'imgsource' property of contact class.

Output on Android and Windows



This template looks cool on all mobile phone devices. But the List looks weird on a Windows desktop and there is no way to set image height in ImageCell. You have to make your own custom cell to overcome this problem. Let's discuss how to make a custom cell (ViewCell). Read the points blow.

ViewCell

ViewCell is the third cell that we use in list. In ViewCell, we can make a custom View of list. That is the View made according to our desire. Let's go through its example.

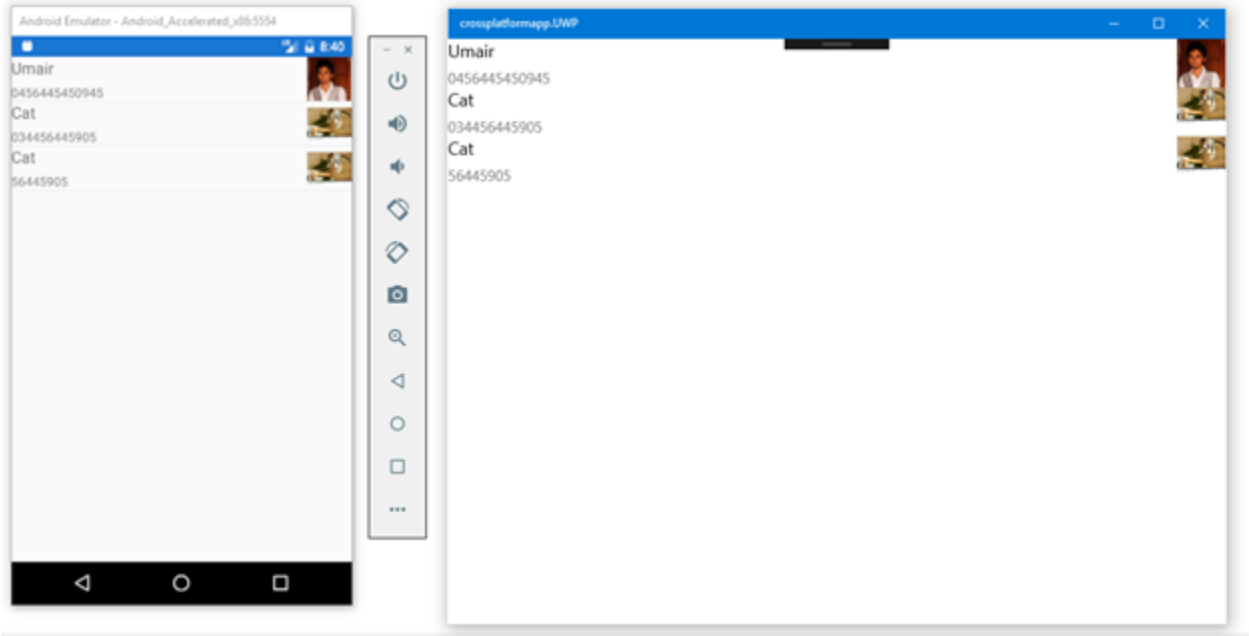
XAML

```
1. <ListView x:Name="lst" HasUnevenRows="True">
2.     <ListView.ItemTemplate>
3.         <DataTemplate>
```

```
4.          <ViewCell>
5.            <StackLayout Orientation="Horizontal">
6.              <StackLayout Orientation="Vertical">
7.                <Label Text="{Binding Name}" Font="18"></Label>
8.                <Label Text="{Binding Num}" TextColor="Gray"></Label> </StackLayout>
9.              <Image Source="{Binding imgsource}" HeightRequest="30" WidthRequest="5
    0" HorizontalOptions="EndAndExpand"></Image>
10.           </StackLayout>
11.         </ViewCell>
12.       </DataTemplate>
13.   </ListView.ItemTemplate>
14. </ListView>
```

Here, you can see that we made two stack layouts with different orientation to make our ViewCell different. We can also set its color, height, width, and other options to make it better.

Output on Android and Windows



Now, give it a try and make your own custom ViewCell.

Functionality of a list view

We will discuss three different functionalities of ListView.

- Pull to Refresh
- Selection
- Context Action

Pull To Refresh

Xamarin provides pull to refresh functionality on all platforms. In this functionality, you will pull your list view down to refresh it. While pulling it down, an activity indicator is shown that can read all of your written code and show the list again.

In an actual example, all the data comes from a remote API or from some other source and refreshes your data. But in this example, we can hard code our list data and on refreshing the list, we can add some more data in it.

Let's implement it.

XAML

```
<ListView x:Name="lst" IsPullToRefreshEnabled="True" Refreshing="lst_Refreshing">
</ListView>
```

You can also make item template, data template, or cells in your list. But for now, I just made a simple List View to understand its functionality.

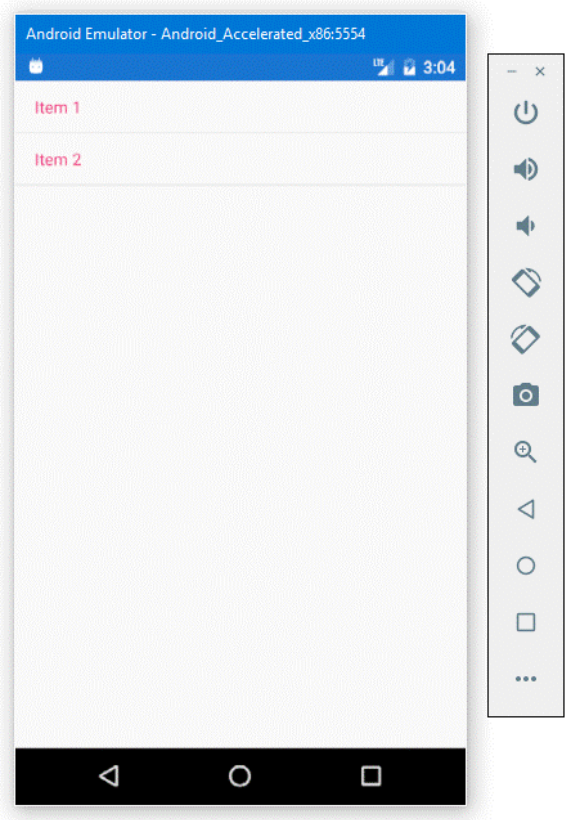
Code

```
public List()
        {
            InitializeComponent();
            lst.ItemsSource = new List<string>() { "Item 1" , "Item 2" };

        }

        private void lst_Refreshing(object sender, EventArgs e)
        {
```

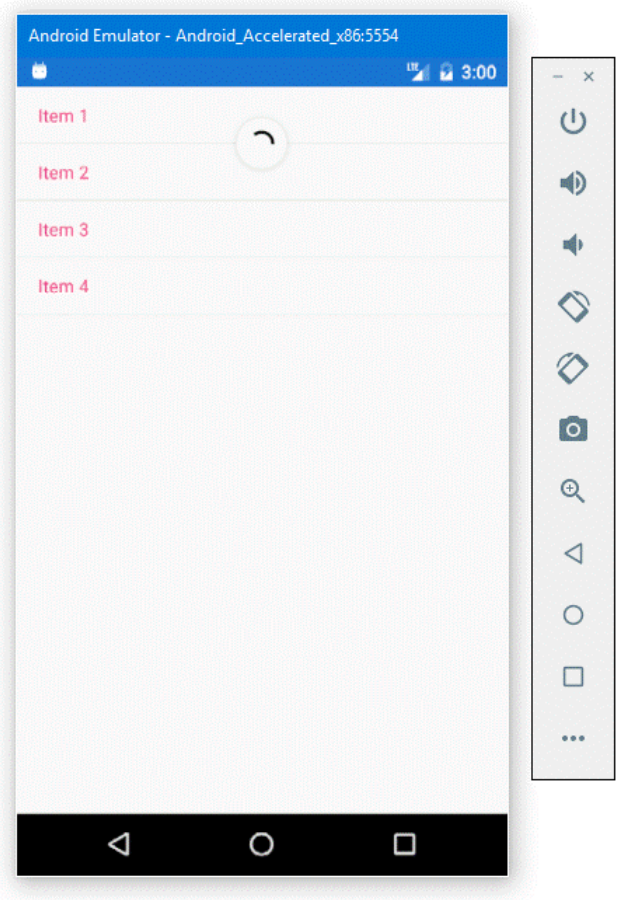
```
10.         lst.ItemsSource = new List<string>() { "Item 1", "Item 2" , "Item 3", "Item 4"};
11.         lst.IsRefreshing = false;
12.
13.     }
```

When the list is initialized, it contains 2 items and after refreshing, it can add 2 more items in it.

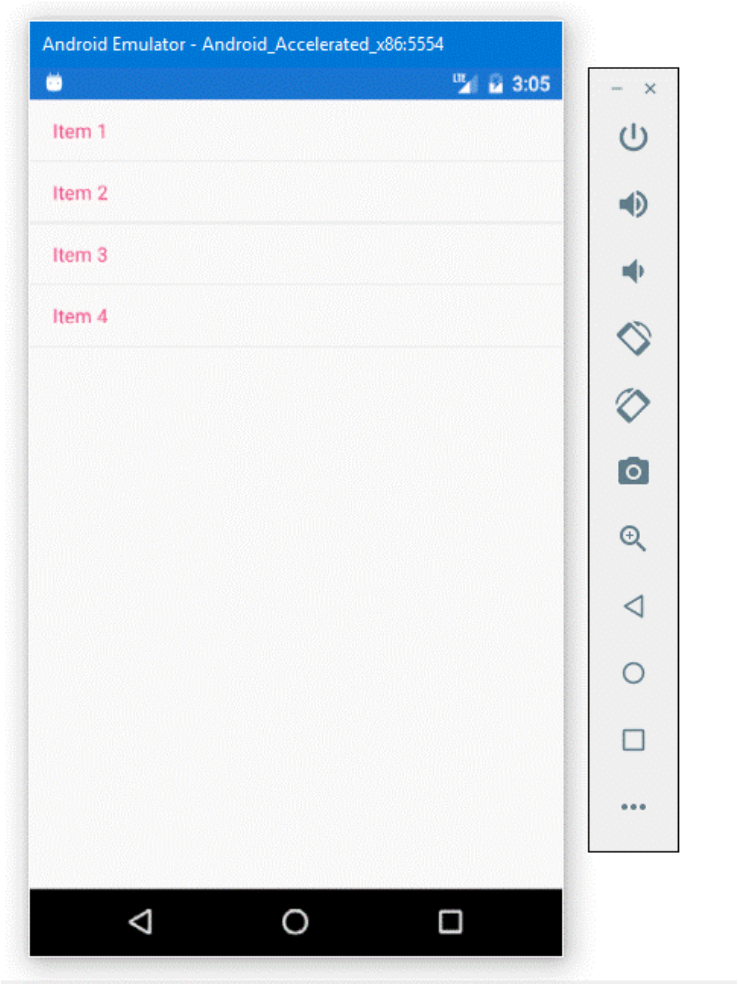
Output on Android



Output - When List is refreshed, you will see an activity indicator.



Output - After refreshing a list.



SELECTION

You can also handle an event when list item is selected. You can move to the next page. Now, any alert or anything you want can be done by this event.

In our example, when user selects a list item, we will show him a display box that contains a message of selected item name in it.

Let's Implement it.

XAML

```
1. <ListView x:Name="lst" ItemSelected="lst_ItemSelected"></ListView>
```

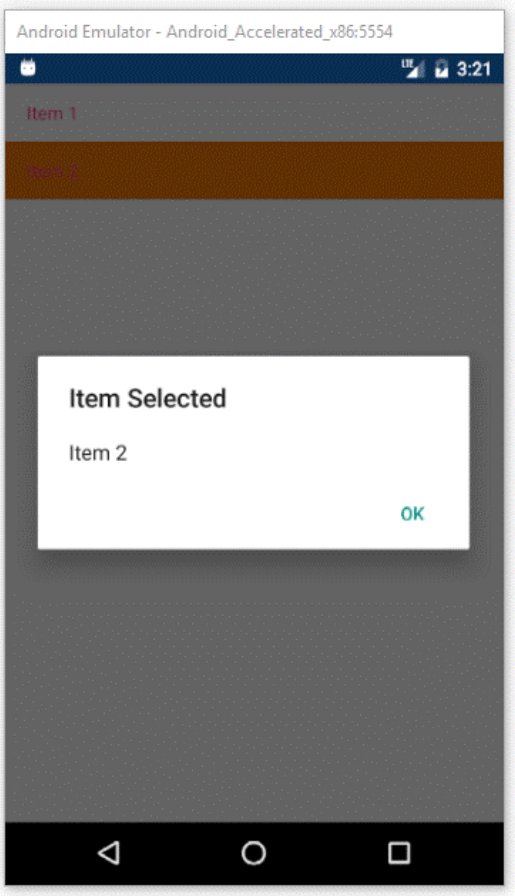
CODE

```
1. public List()
```

```
2.      {
3.          InitializeComponent();
4.          lst.ItemsSource = new List<string>() { "Item 1" , "Item 2" };
5.
6.      }
7.
8.      private void lst_ItemSelected(object sender, SelectedItemChangedEventArgs e)
9.      {
10.         DisplayAlert("Item Selected",e.SelectedItem.ToString(),"ok");
11.     }
```

In selected item event, we can show a display box that contains selected item name.

OUTPUT



Context Action

You can set different context actions in Xamarin.Forms. Context Actions can be shown -

On iOS: Swipe it from right to left.

On Android: when the user performs a long-click (press and hold an item).

You can do any action on it. In this example, we are going to make one context item named "More" and when the More is tapped, the Display box is shown that contains its name.

Implementation

XAML

```
1. <ListView x:Name="lst" ItemSelected="lst_ItemSelected">
2.     <ListView.ItemTemplate>
3.         <DataTemplate>
```

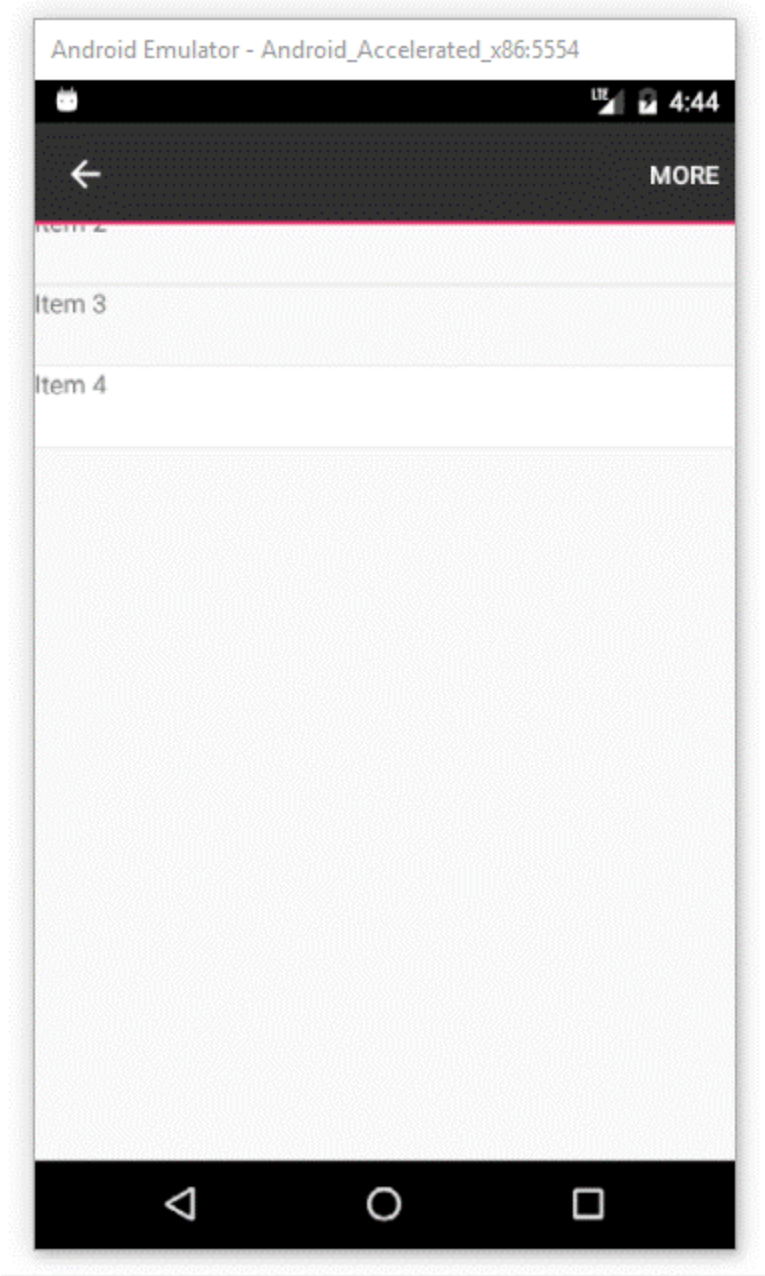
```
4.            <ViewCell>
5.               <StackLayout>
6.                  <Label Text="{Binding .}"></Label>
7.               </StackLayout>
8.               <ViewCell.ContextActions>
9.                  <MenuItem Text="More" x:Name="More" Clicked="More_Clicked" Comma
    ndParameter="{Binding .}"></MenuItem>
10.              </ViewCell.ContextActions>
11.           </ViewCell>
12.        </DataTemplate>
13.     </ListView.ItemTemplate>
14.  </ListView>
```

Here you see that we set a context action to view cell. You may also set context actions to Image cell or Text Cell.

CODE

```
1.     List<string> templist = new List<string>() { "Item 1", "Item 2", "Item 3", "Item 4"};
2.
3.     public List()
4.     {
5.        InitializeComponent();
6.        lst.ItemsSource = templist;
7.     }
8.
9.  private void More_Clicked(object sender, EventArgs e)
10.    {
11.       var name = (sender as MenuItem).CommandParameter as string;
12.       DisplayAlert("Context Action", name, "ok");
13.    }
```

Output - Press and hold an item



Output - When tapped on more

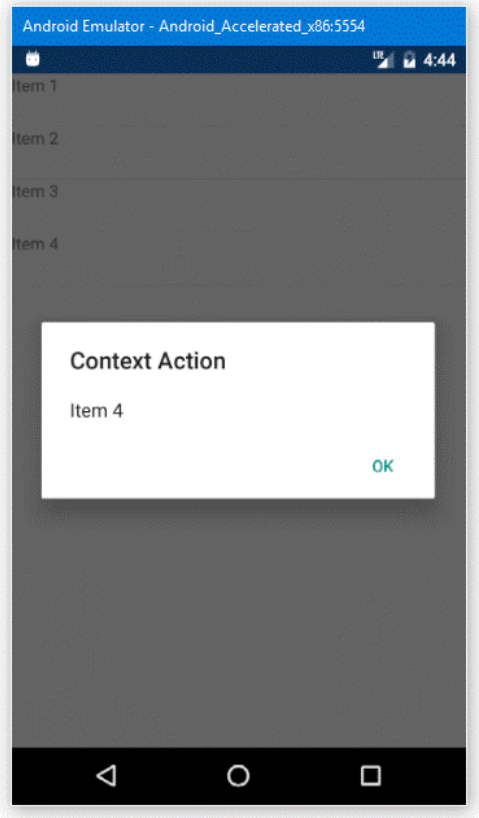
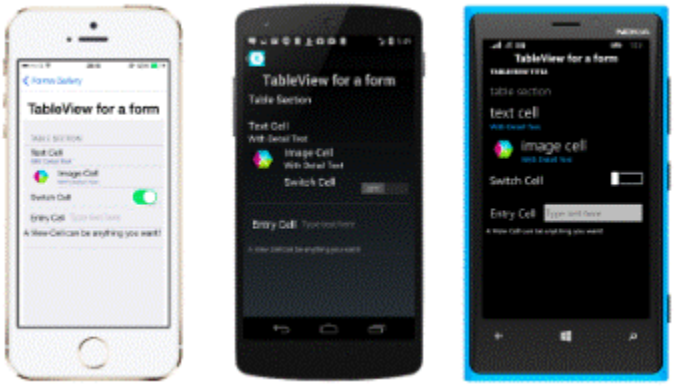


Table View

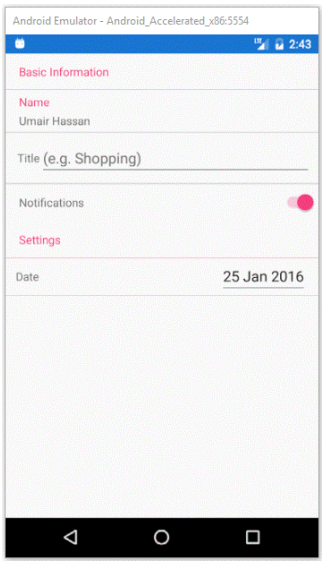
- Table View
- Cells in Table View
- Default templates
- Custom Cells

Table View

It contains rows and columns to place your data in more presentable way.



We are going to make a Table View, which contains groups. It also contains some default cells and custom cells.



We are going to make this view. This Table View contains two groups, where one is of basic information and the second is for settings. Let's start the development.

Development

Let's start by making a single section and giving it a title.

Code

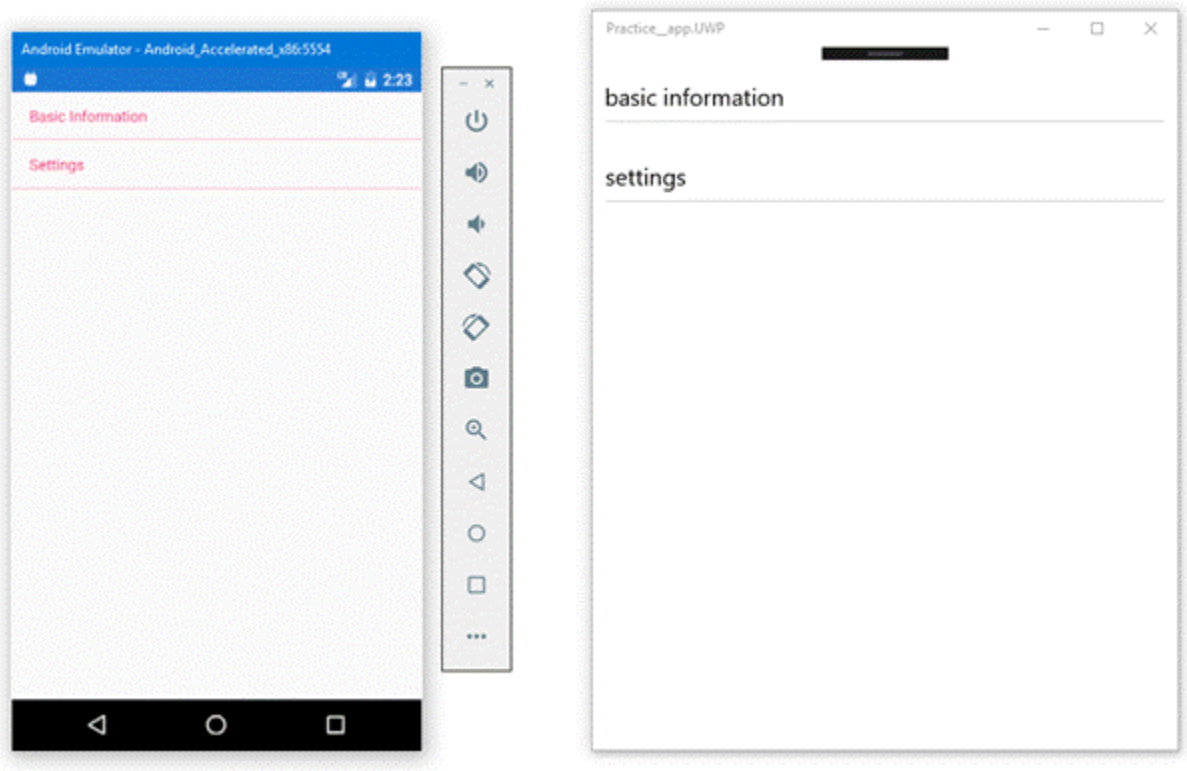
```
<TableView>
    <TableRoot>
        <TableSection  Title="Basic Information">

        </TableSection>
    </TableRoot>
</TableView>
```

Here, you see that we make a Table View tag, which contains table root and table section. We can declare multiple sections in the table and give them a title. Now, make one more section.

Code

```
 <TableView>
    <TableRoot>
        <TableSection  Title="Basic Information">

        </TableSection>

<TableSection  Title="Settings">

        </TableSection>

    </TableRoot>
</TableView>
```

Output on an Android and Windows desktop



Here, you can see that we made 2 sections. Now, add some cells in Table View sections.

Default Templates Cells

Some already made templates are given to make our development easier. We can make an image cell i.e. the cell, which contains an image, a Text cell i.e. a cell, which contains a text and details, an entry cell, which contains an entry to take some input from the user etc. Let's implement some default templates of the cells.

Code

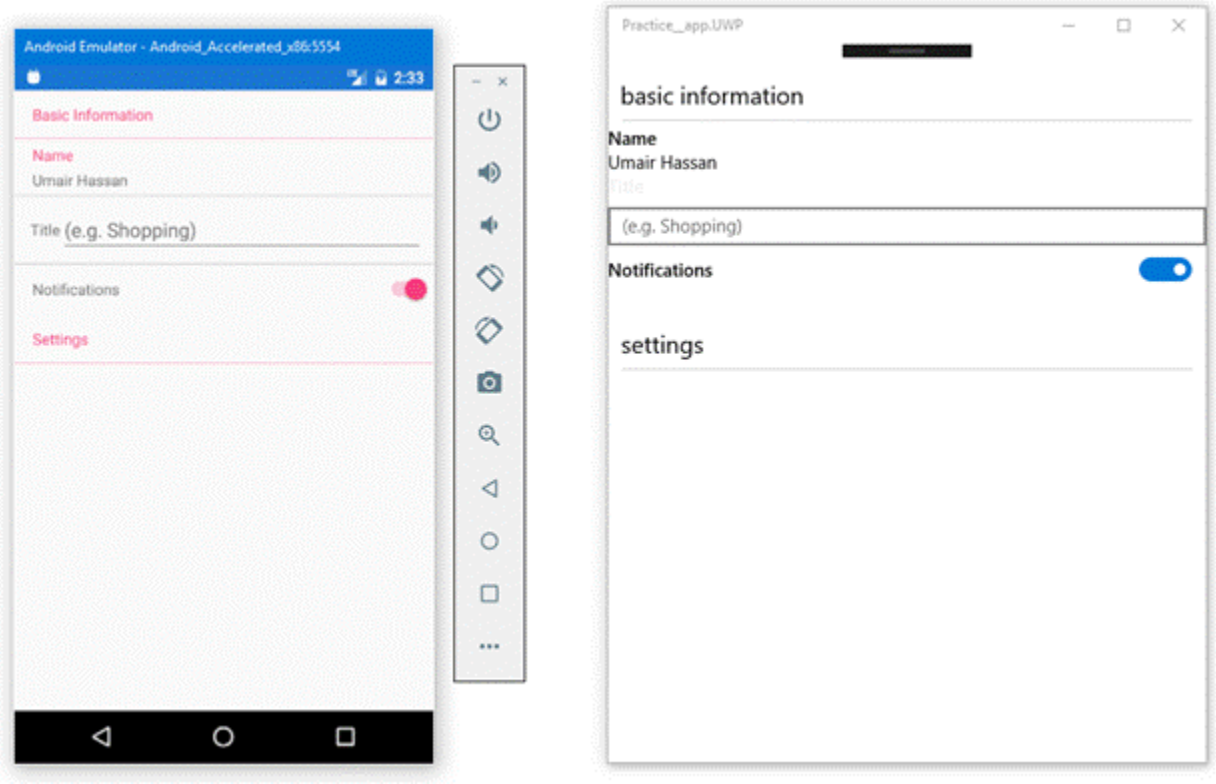
```
1.  <TableView>
2.      <TableRoot>
3.          <TableSection  Title="Basic Information">
4.              <TextCell Text="Name" Detail="Umair Hassan" ></TextCell>
5.              <EntryCell Label="Title" Placeholder="(e.g. Shopping)"></EntryCell>
6.              <SwitchCell Text="Notifications" On="True"></SwitchCell>
7.          </TableSection>
```

```
8.          <TableSection Title="Settings">
9.
10.         </TableSection>
11.     </TableRoot>
12.  </TableView>
```

Here, you see that three cells are used, which are given below.

- *Text cell*
  It contains a text and detail
- *Entry cell*
  It contains a label and Placeholder for an entry.
- *Switch cell*
  Switch cell contains a switch and a notification and we set it to true.

Output



Custom Cell

What if we want to make a cell of our desire? To make a custom cell, we use a tag named 'view cell'. In view cell, we are able to make a custom template. We are free to use any UI element in it. We can also use layout container in our view cell. Let's make a view cell.

Code

```
<ViewCell>
        <StackLayout Orientation="Horizontal" Padding="13,0">
          <Label Text="Date" VerticalOptions="Center"></Label>
          <DatePicker Format="dd MMM yyyy" Date="25 Jan 2016" HorizontalOptions="EndAndExpand"></DatePicker>
        </StackLayout
      </ViewCell>
```

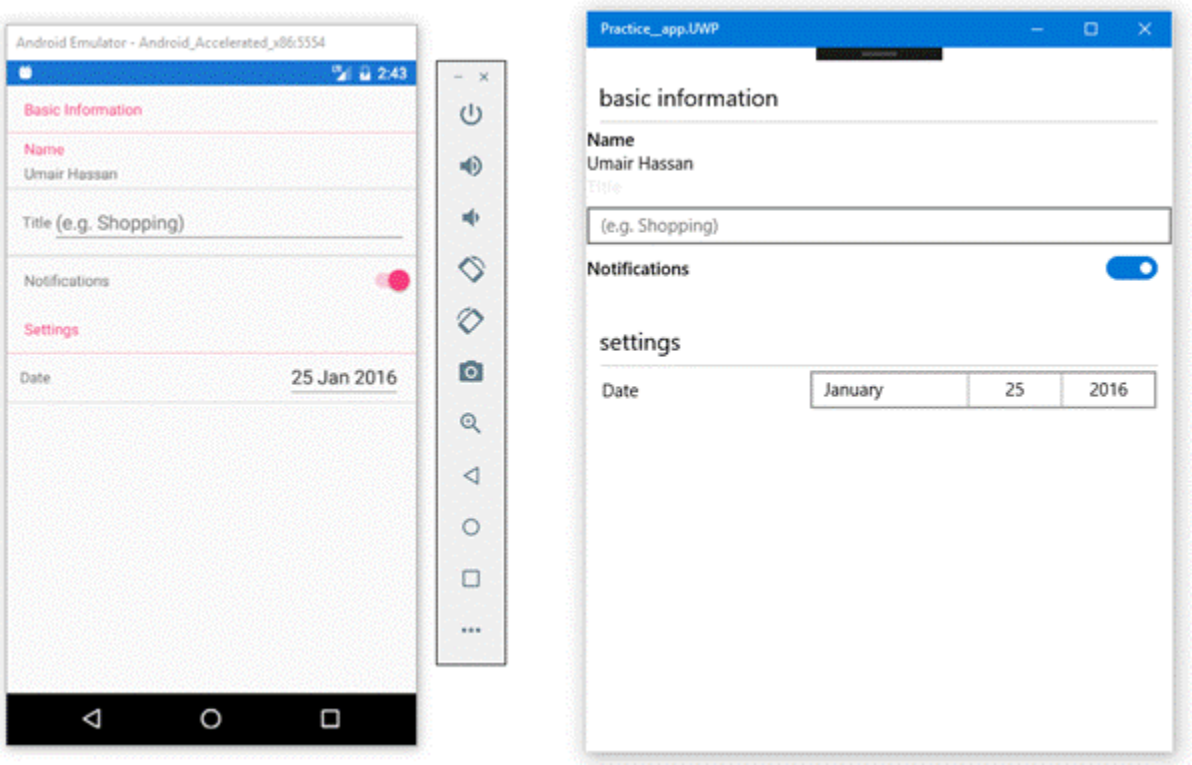
Here, we make a view cell, which contains a stack layout. We can set its orientation and padding, as per our desire and in stack layout, two elements are placed. This view is created, as per our desire.

Full Code

```
<TableView>
  <TableRoot>
    <TableSection  Title="Basic Information">
      <TextCell Text="Name" Detail="Umair Hassan" ></TextCell>
      <EntryCell Label="Title" Placeholder="(e.g. Shopping)"></EntryCell>
      <SwitchCell Text="Notifications" On="True"></SwitchCell>
    </TableSection>
    <TableSection Title="Settings">
      <ViewCell>
        <StackLayout Orientation="Horizontal" Padding="13,0">
          <Label Text="Date" VerticalOptions="Center"></Label>
```

```
12.                 <DatePicker Format="dd MMM yyyy" Date="25 Jan 2016" HorizontalOptions
    ="EndAndExpand"></DatePicker>
13.               </StackLayout>
14.             </ViewCell>
15.         </TableSection>
16.     </TableRoot>
17. </TableView>
```

Output on an Android and Windows desktop



Here, we need to learn to make a custom view cell and we also use cells with already made templates. Now, it's your turn to implement an Image cell.

Note:

This book is created by combining my articles that are written on c-sharpcorner.com. You can visit my profile and learn more.