

# Contents

## Xamarin.Forms

[Get Started](#)

[Requirements](#)

[Build your first app](#)

[Hello, Xamarin.Forms](#)

[Part 1: Quickstart](#)

[Part 2: Deep Dive](#)

[Hello, Xamarin.Forms Multiscreen](#)

[Part 1: Quickstart](#)

[Part 2: Deep Dive](#)

[Introduction to Xamarin.Forms](#)

## XAML

[XAML Basics](#)

[Part 1. Get Started with XAML](#)

[Part 2. Essential XAML Syntax](#)

[Part 3. XAML Markup Extensions](#)

[Part 4. Data Binding Basics](#)

[Part 5. From Data Bindings to MVVM](#)

[XAML Compilation](#)

[XAML Toolbox](#)

[XAML Previewer](#)

[XAML Namespaces](#)

[XAML Markup Extensions](#)

[Consuming XAML Markup Extensions](#)

[Creating XAML Markup Extensions](#)

[Field Modifiers](#)

[Passing Arguments](#)

[Bindable Properties](#)

[Attached Properties](#)

[Resource Dictionaries](#)

[XAML Standard \(Preview\)](#)

[Controls](#)

[Application Fundamentals](#)

[Accessibility](#)

[Automation Properties](#)

[Keyboard Navigation](#)

[App Class](#)

[App Lifecycle](#)

[Behaviors](#)

[Introduction](#)

[Attached Behaviors](#)

[Xamarin.Forms Behaviors](#)

[Reusable Behaviors](#)

[EffectBehavior](#)

[EventToCommandBehavior](#)

[Custom Renderers](#)

[Introduction](#)

[Renderer Base Classes and Native Controls](#)

[Customizing an Entry](#)

[Customizing a ContentPage](#)

[Customizing a Map](#)

[Customizing a Map Pin](#)

[Highlighting a Circular Area on a Map](#)

[Highlighting a Region on a Map](#)

[Highlighting a Route on a Map](#)

[Customizing a ListView](#)

[Customizing a ViewCell](#)

[Implementing a View](#)

[Implementing a HybridWebView](#)

[Implementing a Video Player](#)

[Creating the Platform Video Players](#)

[Playing a Web Video](#)

[Binding Video Sources to the Player](#)

[Loading Application Resource Videos](#)

[Accessing the Device's Video Library](#)

[Custom Video Transport Controls](#)

[Custom Video Positioning](#)

[Data Binding](#)

[Basic Bindings](#)

[Binding Mode](#)

[String Formatting](#)

[Binding Path](#)

[Binding Value Converters](#)

[Binding Fallbacks](#)

[The Command Interface](#)

[Compiled Bindings](#)

[DependencyService](#)

[Introduction](#)

[Implementing Text-to-Speech](#)

[Checking Device Orientation](#)

[Checking Battery Status](#)

[Picking from the Photo Library](#)

[Effects](#)

[Introduction](#)

[Effect Creation](#)

[Passing Parameters](#)

[Parameters as CLR Properties](#)

[Parameters as Attached Properties](#)

[Invoking Events](#)

[Files](#)

[Gestures](#)

[Tap](#)

[Pinch](#)

- Pan
- Swipe
- Localization
  - String and Image Localization
  - Right-to-Left Localization
- Local Databases
- MessagingCenter
- Navigation
  - Hierarchical Navigation
  - TabPage
  - CarouselPage
  - MasterDetailPage
  - Modal Pages
  - Displaying Pop-Ups
- Templates
  - Control Templates
    - Introduction
    - Control Template Creation
    - Template Bindings
  - Data Templates
    - Introduction
    - Data Template Creation
    - Data Template Selection
- Triggers
- User Interface
- Animation
  - Simple Animations
  - Easing Functions
  - Custom Animations
- BoxView
- Button
- Colors

## Controls Reference

- [Pages](#)
- [Layouts](#)
- [Views](#)
- [Cells](#)
- [DataPages](#)
- [Get Started](#)
- [Controls Reference](#)
- [DatePicker](#)
- [Graphics with SkiaSharp](#)
- [Images](#)
- [ImageButton](#)
- [Layouts](#)
- [StackLayout](#)
- [AbsoluteLayout](#)
- [RelativeLayout](#)
- [Grid](#)
- [FlexLayout](#)
- [ScrollView](#)
- [LayoutOptions](#)
- [Margin and Padding](#)
- [Device Orientation](#)
- [Tablet & Desktop](#)
- [Creating a Custom Layout](#)
- [Layout Compression](#)
- [ListView](#)
- [Data Sources](#)
- [Cell Appearance](#)
- [List Appearance](#)
- [Interactivity](#)
- [Performance](#)
- [Maps](#)

[Picker](#)

[Setting a Picker's ItemsSource Property](#)

[Adding Data to a Picker's Items Collection](#)

[Slider](#)

[Stepper](#)

[Styles](#)

[Styling Xamarin.Forms Apps using XAML Styles](#)

[Introduction](#)

[Explicit Styles](#)

[Implicit Styles](#)

[Global Styles](#)

[Style Inheritance](#)

[Dynamic Styles](#)

[Device Styles](#)

[Styling Xamarin.Forms Apps using Cascading Style Sheets \(CSS\)](#)

[TableView](#)

[Text](#)

[Label](#)

[Entry](#)

[Editor](#)

[Fonts](#)

[Styles](#)

[Themes](#)

[Light Theme](#)

[Dark Theme](#)

[Creating a Custom Theme](#)

[TimePicker](#)

[Visual State Manager](#)

[WebView](#)

[Platform Features](#)

[Android](#)

[AppCompat & Material Design](#)

[Application Indexing and Deep Linking](#)

[Device Class](#)

[iOS](#)

[Formatting](#)

[GTK#](#)

[Mac](#)

[Native Forms](#)

[Native Views](#)

[Native Views in XAML](#)

[Native Views in C#](#)

[Platform Specifics](#)

[Consuming Platform-Specifics](#)

[iOS](#)

[Android](#)

[Windows](#)

[Creating Platform-Specifics](#)

[Plugins](#)

[Tizen](#)

[Windows](#)

[Setup](#)

[WPF](#)

[Xamarin.Essentials](#)

[Get Started](#)

[Accelerometer](#)

[App Information](#)

[Barometer](#)

[Battery](#)

[Clipboard](#)

[Compass](#)

[Connectivity](#)

[Data Transfer](#)

[Device Display Information](#)

[Device Information](#)

[Email](#)

[File System Helpers](#)

[Flashlight](#)

[Geocoding](#)

[Geolocation](#)

[Gyroscope](#)

[Launcher](#)

[Magnetometer](#)

[Main Thread](#)

[Maps](#)

[Open Browser](#)

[Orientation Sensor](#)

[Phone Dialer](#)

[Power](#)

[Preferences](#)

[Screen Lock](#)

[Secure Storage](#)

[SMS](#)

[Text-to-Speech](#)

[Version Tracking](#)

[Vibrate](#)

[Troubleshooting](#)

[Data & Cloud Services](#)

[Understanding the Sample](#)

[Consuming Web Services](#)

[ASMX](#)

[WCF](#)

[REST](#)

[Azure Mobile Apps](#)

[Authenticating Access to Web Services](#)

[REST](#)

OAuth  
Azure Mobile Apps  
Azure Active Directory B2C  
Integrating Azure Active Directory B2C with Azure Mobile Apps  
Synchronizing Data with Web Services  
    Azure Mobile Apps  
    Sending Push Notifications  
    Azure  
    Storing Files in the Cloud  
    Azure Storage  
    Searching Data in the Cloud  
    Azure Search  
    Serverless Applications  
    Azure Functions  
    Storing Data in a Document Database  
        Consuming an Azure Cosmos DB Document Database  
        Authenticating Users with an Azure Cosmos DB Document Database  
    Adding Intelligence with Cognitive Services  
        Speech Recognition  
        Spell Check  
        Text Translation  
        Emotion Recognition  
Deployment & Testing  
    Performance  
    Xamarin.UITest and Test Cloud  
Advanced Concepts and Internals  
    Fast Renderers  
    .NET Standard  
    Dependency Resolution  
Troubleshooting  
Frequently Asked Questions  
    Can I update the Xamarin.Forms default template to a newer NuGet package?

[Why doesn't the Visual Studio XAML designer work for Xamarin.Forms XAML files?](#)

[Android build error: The "LinkAssemblies" task failed unexpectedly](#)

[Why does my Xamarin.Forms.Maps Android project fail with COMPILETODALVIK : UNEXPECTED TOP-LEVEL ERROR?](#)

[Release Notes](#)

[Samples](#)

[Creating Mobile Apps with Xamarin.Forms Book](#)

[Enterprise Application Patterns eBook](#)

[SkiaSharp Graphics in Xamarin.Forms](#)

# Get Started with Xamarin.Forms

10/10/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms is a cross-platform UI toolkit that allows developers to efficiently create native user interface layouts that can be shared across iOS, Android, and Universal Windows Platform apps. This series introduces the basics of Xamarin.Forms development and covers building multi-platform and multi-screen applications.*

For an overview of the installation and setup practices that apply to cross-platform development, see [Xamarin.Forms Requirements](#) and [Installation](#).

[Build your first app](#)

## Requirements

Overview of the platform requirements for Xamarin.Forms-developed apps, and the minimum system requirements for developing with Xamarin.Forms in Visual Studio for Mac and Visual Studio.

## Build your first app

Watch a video and follow along with step-by-step instructions to build and test your first Xamarin.Forms app.

## Hello, Xamarin.Forms

This guide provides an introduction to developing a Xamarin.Forms application using Visual Studio for Mac or Visual Studio. Topics covered include the tools, concepts, and steps required to build and deploy a Xamarin.Forms application.

## Hello, Xamarin.Forms Multiscreen

This guide extends the previously created application by introducing navigation to a second page. Topics covered include data binding and performing navigation.

## Introduction To Xamarin.Forms

This article discusses some of the key concepts for developing applications using Xamarin.Forms, including [Views and Layouts](#), the [ListView](#) control, [Data Binding](#) and [Navigation](#).

## Get Started with Xamarin University

**[Building Your First Xamarin.Forms App with Xamarin for Visual Studio, by Xamarin University](#)**

## Related Links

- [Free Self-Guided Learning \(video\)](#)
- [Getting Started with Xamarin \(video\)](#)

# Xamarin.Forms Requirements

10/17/2018 • 2 minutes to read • [Edit Online](#)

*Platform and development system requirements for Xamarin.Forms.*

Refer to the [Installation](#) article for an overview of installation and setup practices that apply across platforms.

## Target platforms

Xamarin.Forms applications can be written for the following operating systems:

- iOS 8 or higher
- Android 4.4 (API 19) or higher ([more details](#))
- Windows 10 Universal Windows Platform ([more details](#))

It is assumed that developers have familiarity with [.NET Standard](#) and [Shared Projects](#).

### Additional platform support

The status of these platforms is available on the [Xamarin.Forms GitHub](#):

- Samsung Tizen
- macOS
- GTK#
- WPF

### Platforms from earlier versions

These platforms are not supported when using Xamarin.Forms 3.0:

- *Windows 8.1 / Windows Phone 8.1 WinRT*
- *Windows Phone 8 Silverlight*

### Android

You should have the latest Android SDK Tools and Android API platform installed. You can update to the latest versions using the [Android SDK Manager](#).

Additionally, the target/compile version for Android projects **must** be set to *Use latest installed platform*. However the minimum version can be set to API 19 so you can continue to support devices that use Android 4.4 and newer. These values are set in the **Project Options**:

- [Visual Studio](#)
- [Visual Studio for Mac](#)

### Project Options > Application > Application Properties



## Development system requirements

Xamarin.Forms apps can be developed on macOS and Windows. However, Windows and Visual Studio are required to produce Windows versions of the app.

## Mac System requirements

You can use Visual Studio for Mac to develop Xamarin.Forms apps on OS X El Capitan (10.11) or newer. To develop iOS apps, we recommend having at least the iOS 10 SDK and Xcode 8 installed.

### NOTE

Windows apps cannot be developed on macOS.

## Windows system requirements

Xamarin.Forms apps for iOS and Android can be built on any Windows installation that supports Xamarin development. This requires Visual Studio 2017 or newer running on Windows 7 or higher. A networked Mac is required for iOS development.

### Universal Windows Platform (UWP)

Developing Xamarin.Forms apps for UWP requires:

- Windows 10 (Fall Creators Update recommended)
- Visual Studio 2017
- [Windows 10 SDK](#)

UWP projects are included in Xamarin.Forms solutions created in Visual Studio 2017, but not solutions created in Visual Studio for Mac. You can [add a Universal Windows Platform \(UWP\) App](#) to an existing Xamarin.Forms solution at any time.

# Build your first Xamarin.Forms App

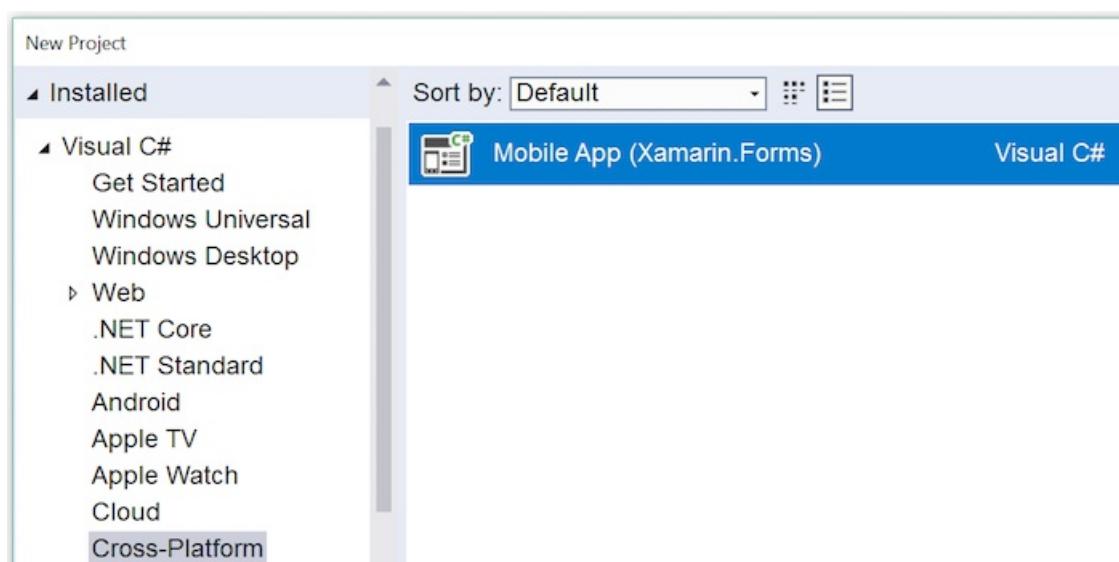
9/28/2018 • 2 minutes to read • [Edit Online](#)

Watch this video and follow along to create your first mobile app with Xamarin.Forms.

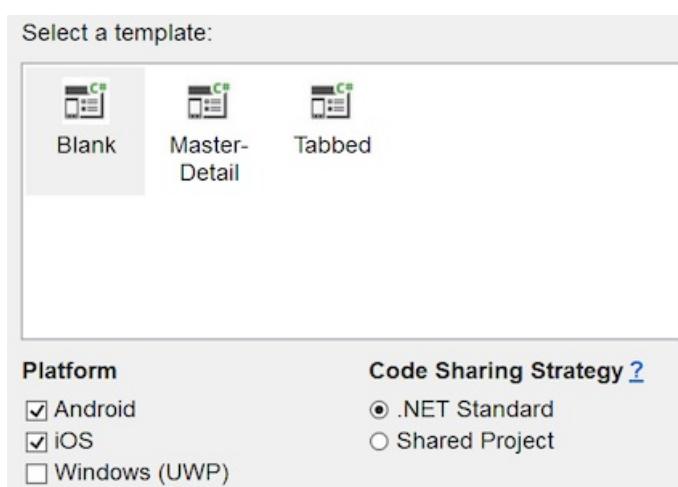
## Step-by-step instructions for Windows

Follow these steps along with the video above:

1. Choose **File > New > Project...** or press the **Create new project...** button, then select **Visual C# > Cross-Platform > Mobile App (Xamarin.Forms)**:



2. Ensure **Android** and **iOS** are selected, with **.NET Standard** code sharing:



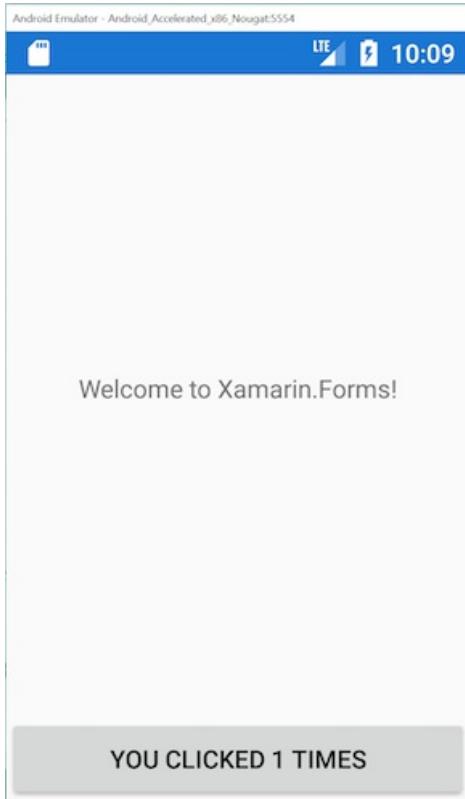
3. Wait until the NuGet packages are restored (a "Restore completed" message will appear in the status bar).
4. Launch Android emulator by pressing the debug button (or the **Debug > Start Debugging** menu item).
5. Edit **MainPage.xaml**, adding this XAML before the end of the `</StackPanel>`:

```
<Button Text="Click Me" Clicked="Button_Clicked" />
```

6. Edit **MainPage.xaml.cs**, adding this code to the end of the class:

```
int count = 0;
void Button_Clicked(object sender, System.EventArgs e)
{
    count++;
    ((Button)sender).Text = $"You clicked {count} times.";
}
```

7. Debug the app on Android:



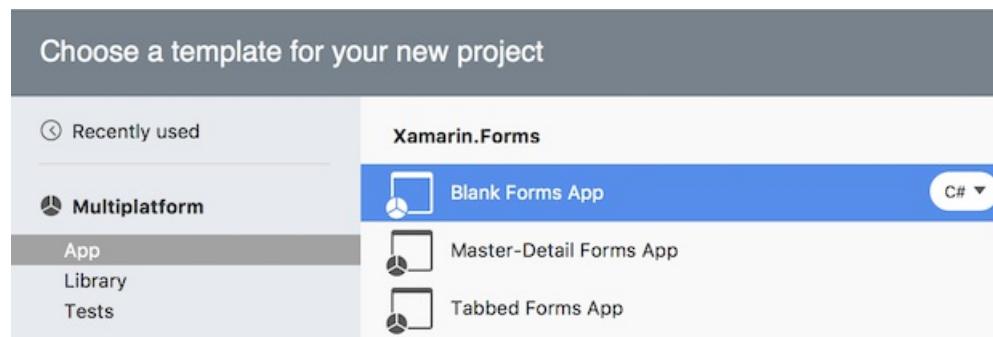
**TIP**

It is possible to build and debug the iOS app from Visual Studio with a networked Mac computer. Refer to the [setup instructions](#) for more information.

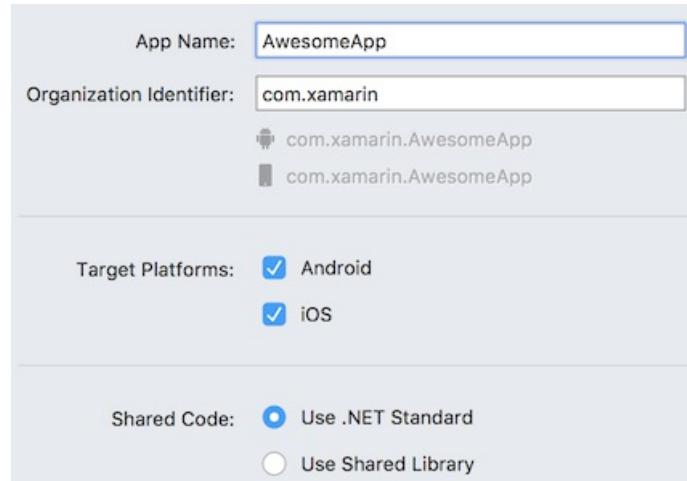
## Step-by-step instructions for Mac

Follow these steps along with the video above:

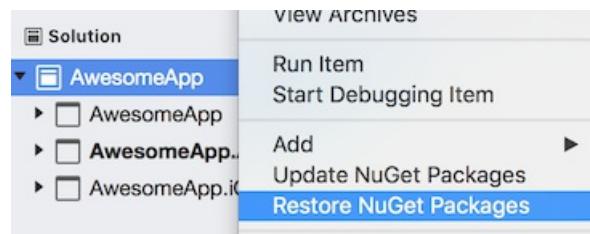
1. Choose **File > New Solution...** or press the **New Project...** button, then select **Multiplatform > App > Blank Forms App:**



2. Ensure **Android** and **iOS** are selected, with **.NET Standard** code sharing:



3. Restore NuGet packages, by right-clicking on the solution:



4. Launch Android emulator by pressing the debug button (or **Run > Start Debugging**).

5. Edit **MainPage.xaml**, adding this XAML before the end of the `</StackPanel>`:

```
<Button Text="Click Me" Clicked="Handle_Clicked" />
```

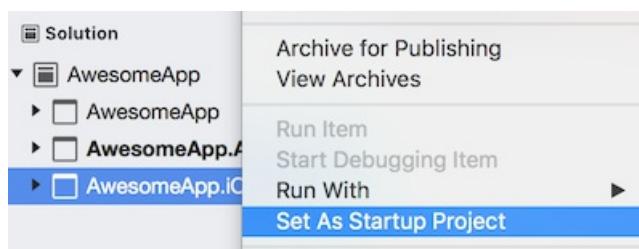
6. Edit **MainPage.xaml.cs**, adding this code to the end of the class:

```
int count = 0;
void Handle_Clicked(object sender, System.EventArgs e)
{
    count++;
    ((Button)sender).Text = $"You clicked {count} times.";
}
```

7. Debug the app on Android:



8. Right-click to set iOS to the **Startup Project**:



9. Debug the app on iOS:



You can download the completed code from the [samples gallery](#) or view it on [GitHub](#).

## Next Steps

- [Hello, Xamarin.Forms](#) – Build a more functional app.
- [Hello, Xamarin.Forms Multi-screen](#) – Build an app that navigates between two screens.
- [Xamarin.Forms Samples](#) – Download and run code examples and sample apps.
- [Introduction to Xamarin.Forms](#) – Beginners guide to Xamarin.Forms.
- [Creating Mobile Apps ebook](#) – In-depth chapters that teach Xamarin.Forms development, available as a PDF and including hundreds of additional samples.

# Hello, Xamarin.Forms

9/26/2018 • 2 minutes to read • [Edit Online](#)

*This guide provides an introduction to developing a Xamarin.Forms application using Visual Studio for Mac or Visual Studio, and to the fundamentals of application development using Xamarin.Forms. Topics covered include the tools, concepts, and steps required to build and deploy a Xamarin.Forms application.*

Start by reviewing the [Xamarin.Forms System Requirements](#).

## Part 1: Quickstart

The first part of this guide demonstrates how to create an application that translates an alphanumeric phone number entered by the user into a numeric phone number, and then calls that number.

## Part 2: Deep Dive

The second part of this guide reviews what has been built, to develop an understanding of the fundamentals of how Xamarin.Forms applications work.

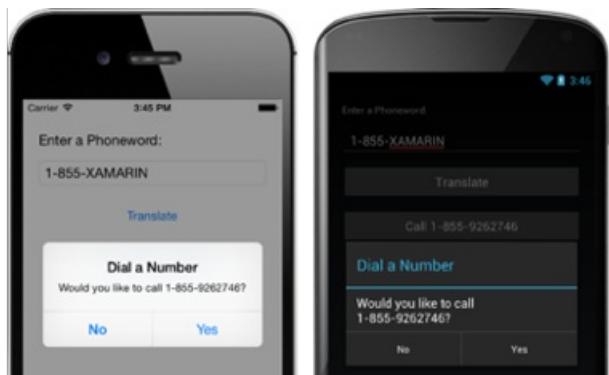
## Related Links

- [Introduction to Xamarin.Forms](#)
- [Debugging in Visual Studio](#)
- [Visual Studio for Mac Recipes - Debugging](#)
- [Free Self-Guided Learning \(video\)](#)
- [Getting Started with Xamarin \(video\)](#)

# Xamarin.Forms Quickstart

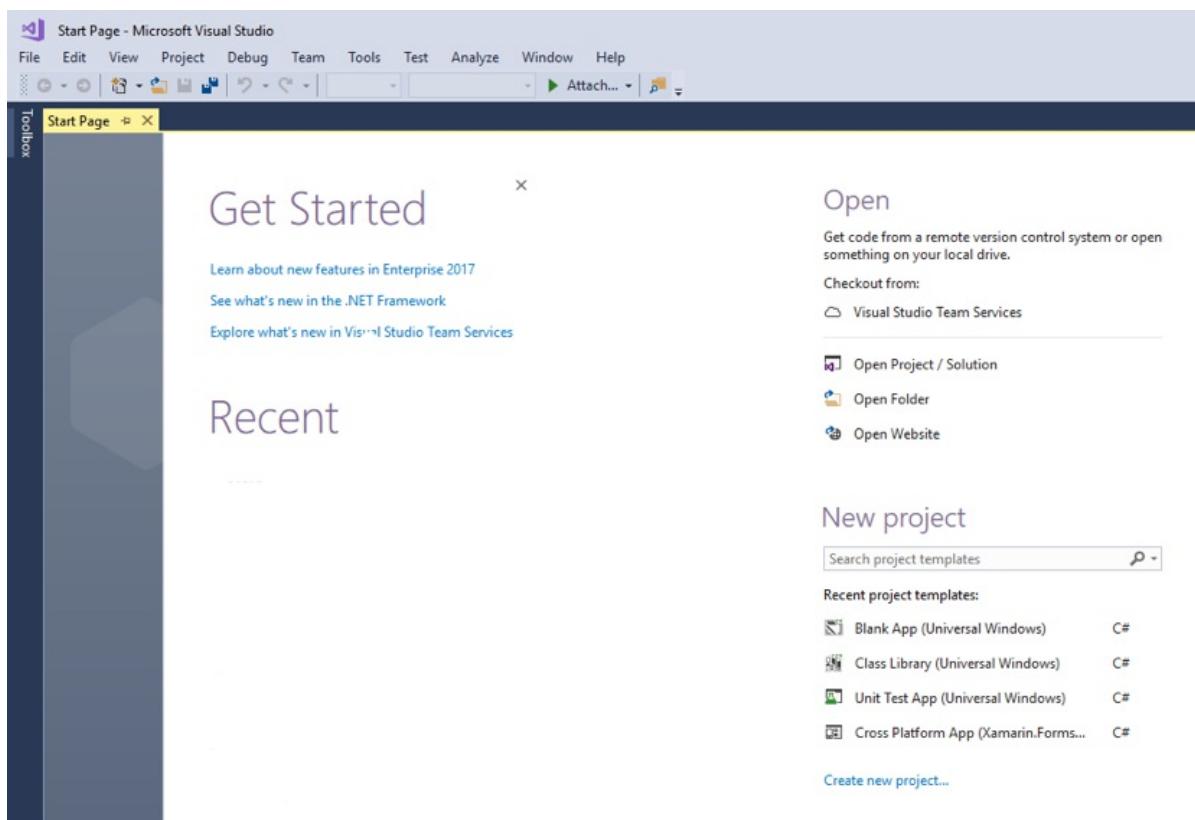
10/25/2018 • 13 minutes to read • [Edit Online](#)

This walkthrough demonstrates how to create an application that translates an alphanumeric phone number entered by the user into a numeric phone number, and that calls the number. The final application is shown below:



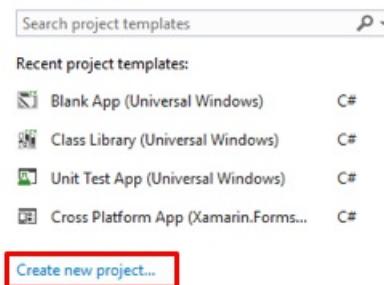
## Get started with Visual Studio

1. In the **Start** screen, launch Visual Studio. This opens the start page:

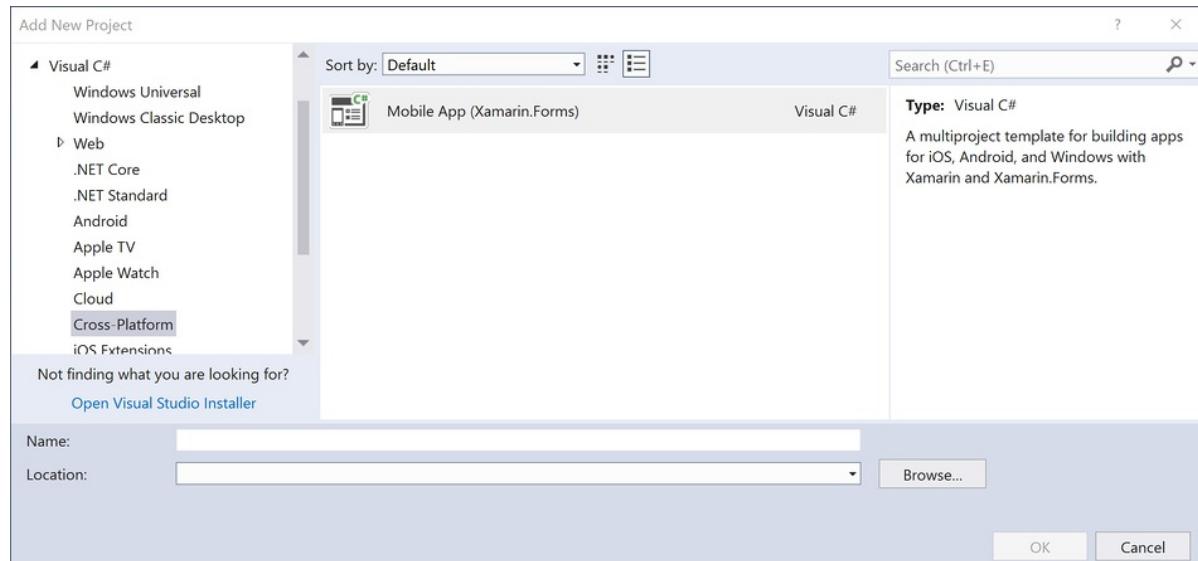


2. In Visual Studio, click **Create new project...** to create a new project:

## New project



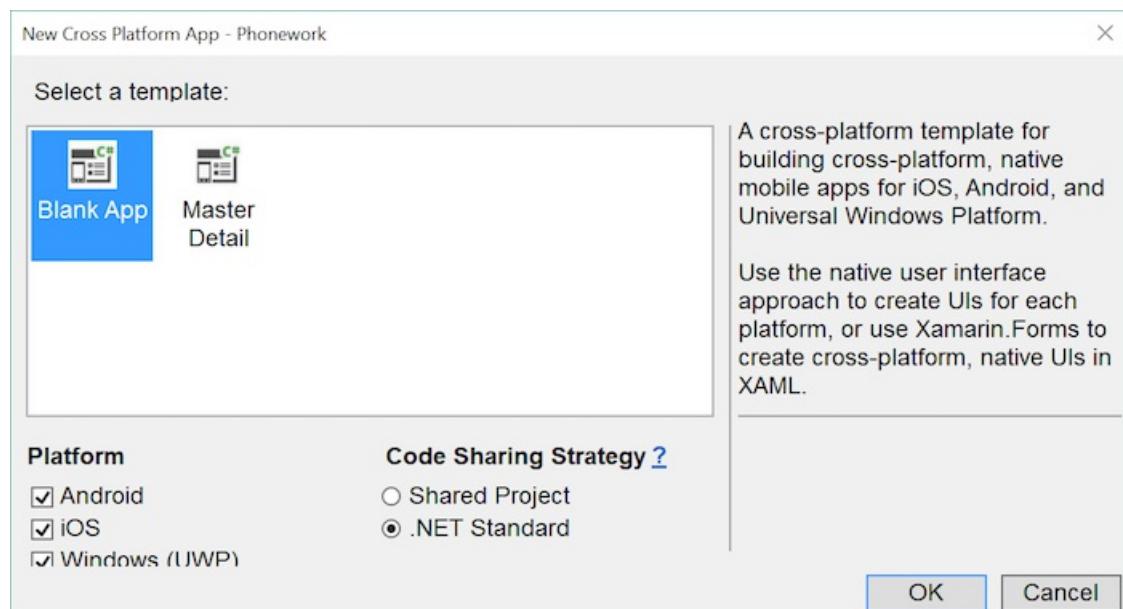
3. In the **New Project** dialog, click **Cross-Platform**, select the **Mobile App (Xamarin.Forms)** template, set the Name to **Phoneword**, choose a suitable location for the project and click the **OK** button:



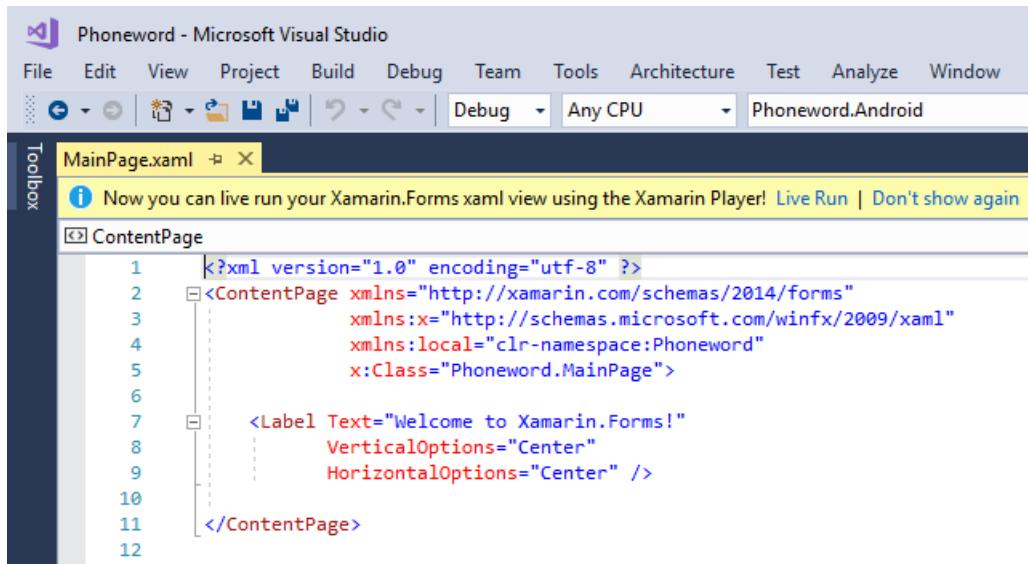
### NOTE

The C# and XAML snippets in this quickstart require the solution be named **Phoneword**. Using a different solution name will result in numerous build errors when you copy code from these instructions into the projects.

4. In the **New Cross Platform App** dialog, click **Blank App**, select **.NET Standard** as the Code Sharing Strategy, and click the **OK** button:



5. In **Solution Explorer**, in the **Phoneword** project, double-click **MainPage.xaml** to open it:



6. In **MainPage.xaml**, remove all of the template code and replace it with the following code. This code declaratively defines the user interface for the page:

```
<?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="Phoneword.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
        <Label Text="Enter a Phoneword:" />
        <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
        <Button Text="Translate" Clicked="OnTranslate" />
        <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>
```

Save the changes to **MainPage.xaml** by pressing **CTRL+S**, and close the file.

7. In **Solution Explorer**, expand **MainPage.xaml** and double-click **MainPage.xaml.cs** to open it:

The screenshot shows the Microsoft Visual Studio interface with the title bar "Phoneword - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, and Architecture. The toolbar has icons for back, forward, search, and file operations. The status bar at the bottom shows "Debug Any CPU". The code editor window is titled "MainPage.xaml.cs" and contains the following C# code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using Xamarin.Forms;
7
8  namespace Phoneword
9  {
10    public partial class MainPage : ContentPage
11    {
12      public MainPage()
13      {
14        InitializeComponent();
15      }
16    }
17  }
18
```

8. In **MainPage.xaml.cs**, remove all of the template code and replace it with the following code. The **OnTranslate** and **OnCall** methods will be executed in response to the **Translate** and **Call** buttons being clicked in the user interface, respectively:

```

using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        string translatedNumber;

        public MainPage ()
        {
            InitializeComponent ();
        }

        void OnTranslate (object sender, EventArgs e)
        {
            translatedNumber = PhonewordTranslator.ToNumber (phoneNumberText.Text);
            if (!string.IsNullOrWhiteSpace (translatedNumber)) {
                callButton.IsEnabled = true;
                callButton.Text = "Call " + translatedNumber;
            } else {
                callButton.IsEnabled = false;
                callButton.Text = "Call";
            }
        }

        async void OnCall (object sender, EventArgs e)
        {
            if (await this.DisplayAlert (
                "Dial a Number",
                "Would you like to call " + translatedNumber + "?",
                "Yes",
                "No")) {
                var dialer = DependencyService.Get<IDialer> ();
                if (dialer != null)
                    dialer.Dial (translatedNumber);
            }
        }
    }
}

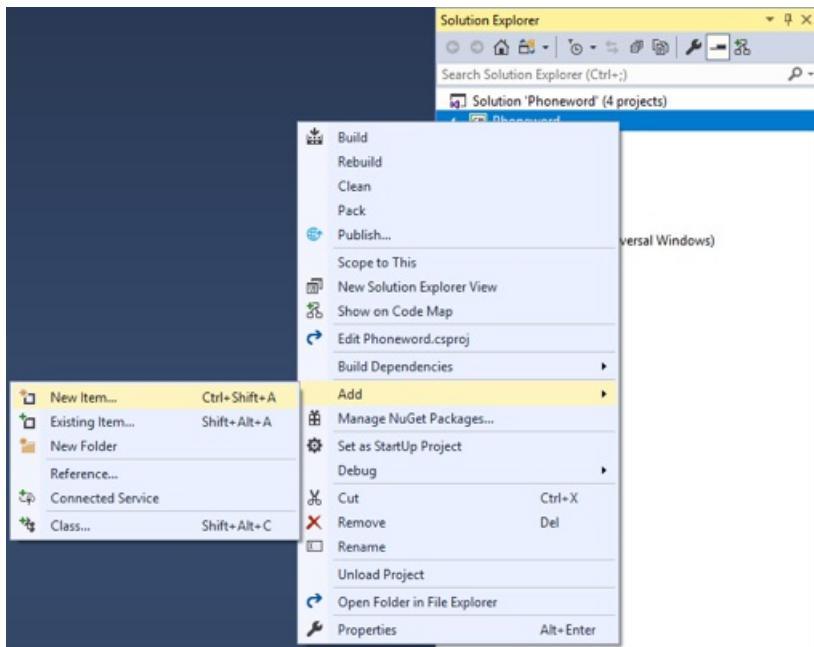
```

#### NOTE

Attempting to build the application at this point will result in errors that will be fixed later.

Save the changes to **MainPage.xaml.cs** by pressing **CTRL+S**, and close the file.

9. In **Solution Explorer**, right click on the **Phoneword** project and select **Add > New Item...**:



10. In the **Add New Item** dialog, select **Visual C# > Code > Class**, name the new file **PhoneTranslator**, and click the **Add** button:



11. In **PhoneTranslator.cs**, remove all of the template code and replace it with the following code. This code will translate a phone word to a phone number:

```

using System.Text;

namespace Phoneword
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return null;

            raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                    newNumber.Append(c);
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                    // Bad character?
                    else
                        return null;
                }
            }
            return newNumber.ToString();
        }

        static bool Contains(this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }

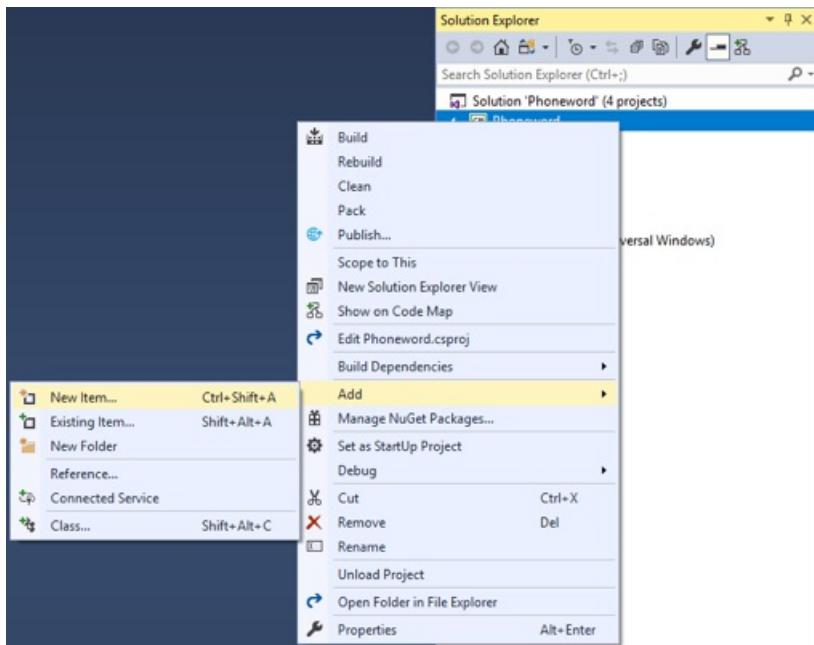
        static readonly string[] digits = {
            "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ"
        };

        static int? TranslateToNumber(char c)
        {
            for (int i = 0; i < digits.Length; i++)
            {
                if (digits[i].Contains(c))
                    return 2 + i;
            }
            return null;
        }
    }
}

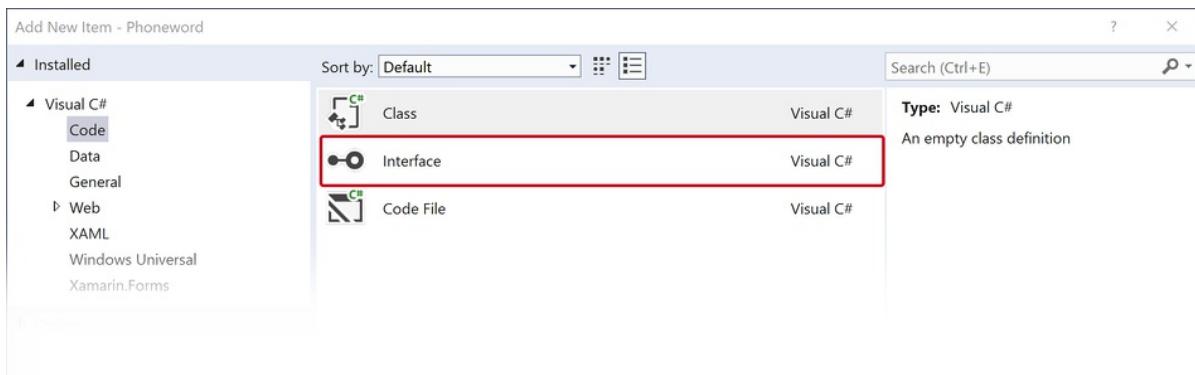
```

Save the changes to **PhoneTranslator.cs** by pressing **CTRL+S**, and close the file.

12. In **Solution Explorer**, right click on the **Phoneword** project and select **Add > New Item...**:



- In the **Add New Item** dialog, select **Visual C# > Code > Interface**, name the new file **IDialer**, and click the **Add** button:



- In **IDialer.cs**, remove all of the template code and replace it with the following code. This code defines a **Dial** method that must be implemented on each platform to dial a translated phone number:

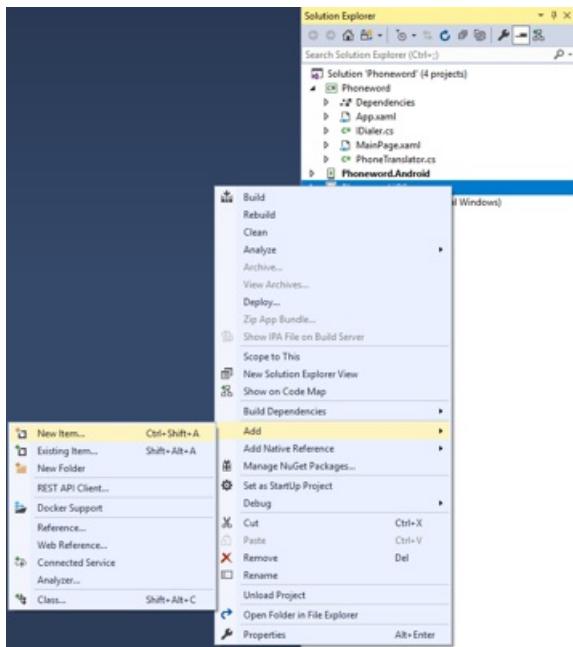
```
namespace Phoneword
{
    public interface IDialer
    {
        bool Dial(string number);
    }
}
```

Save the changes to **IDialer.cs** by pressing **CTRL+S**, and close the file.

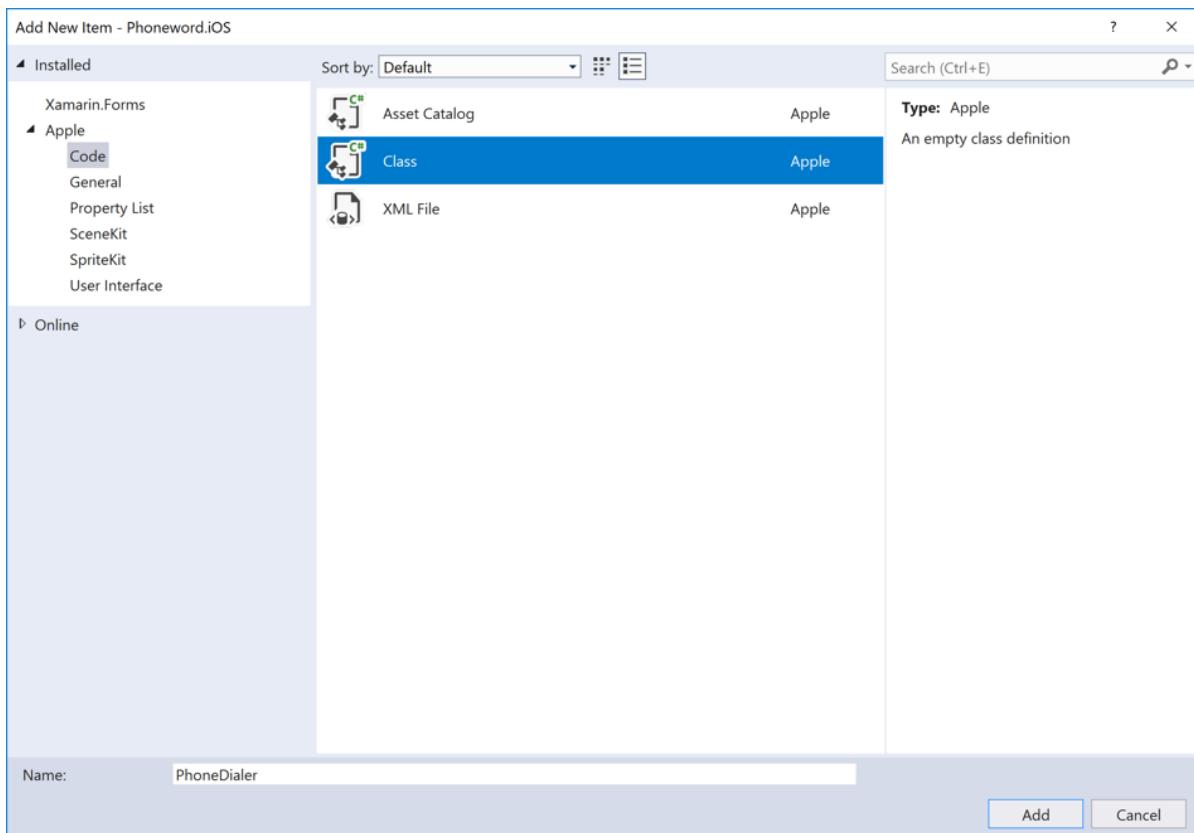
#### NOTE

The common code for the application is now complete. Platform-specific phone dialer code will now be implemented as a [DependencyService](#).

- In **Solution Explorer**, right click on the **Phoneword.iOS** project and select **Add > New Item...**:



16. In the **Add New Item** dialog, select **Apple > Code > Class**, name the new file **PhoneDialer**, and click the **Add** button:



17. In **PhoneDialer.cs**, remove all of the template code and replace it with the following code. This code creates the `Dial` method that will be used on the iOS platform to dial a translated phone number:

```

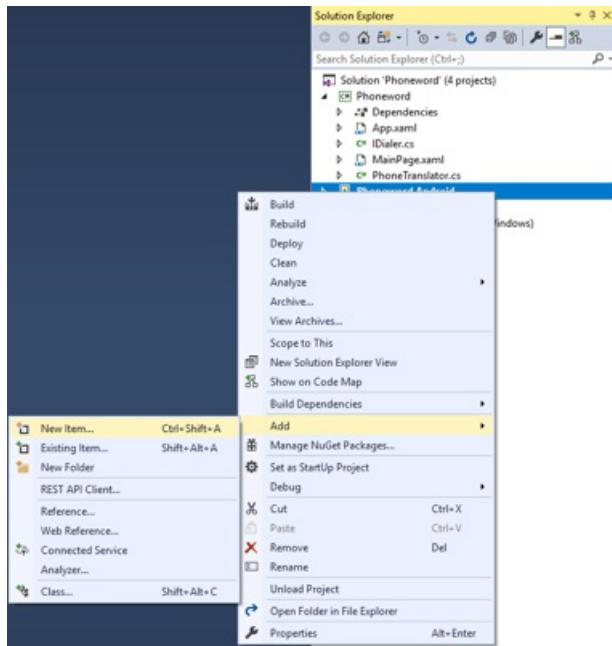
using Foundation;
using Phoneword.iOS;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.iOS
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            return UIApplication.SharedApplication.OpenUrl (
                new NSUrl ("tel:" + number));
        }
    }
}

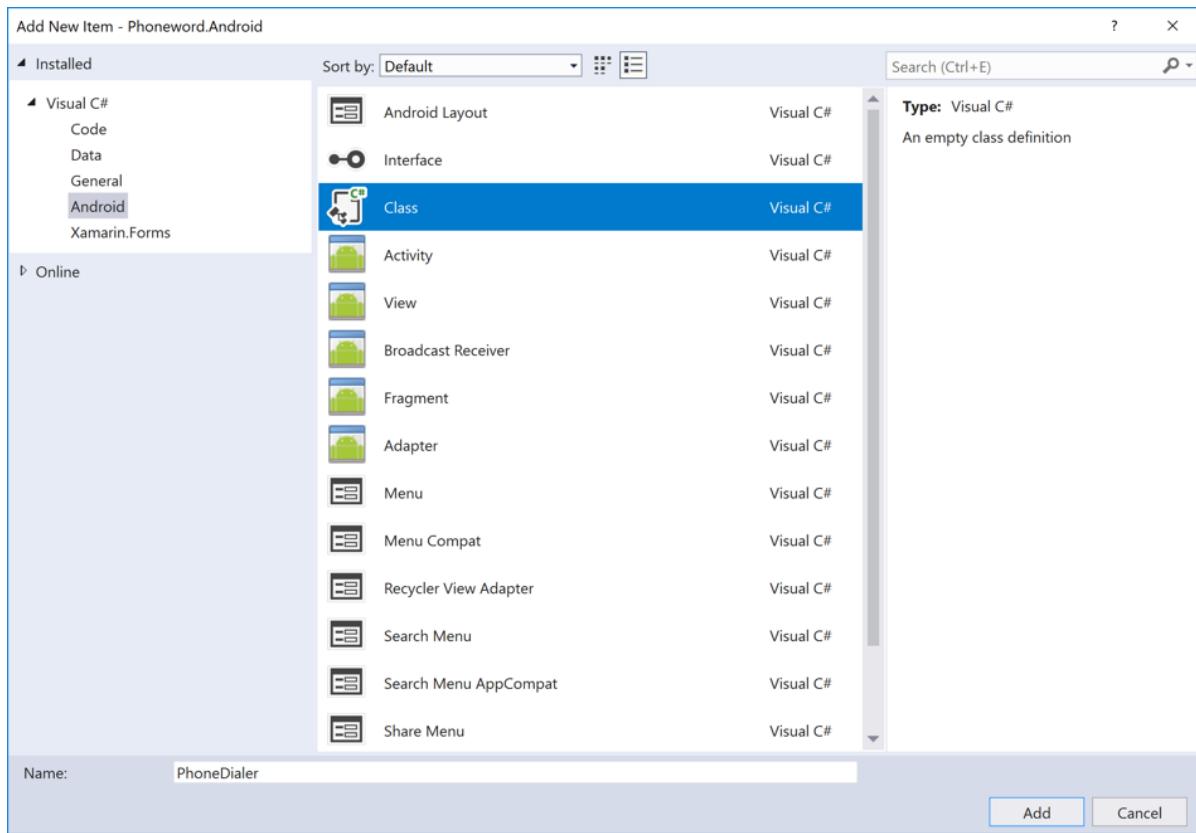
```

Save the changes to **PhoneDialer.cs** by pressing **CTRL+S**, and close the file.

18. In **Solution Explorer**, right click on the **Phoneword.Android** project and select **Add > New Item...**:



19. In the **Add New Item** dialog, select **Visual C# > Android > Class**, name the new file **PhoneDialer**, and click the **Add** button:



20. In **PhoneDialer.cs**, remove all of the template code and replace it with the following code. This code creates the `Dial` method that will be used on the Android platform to dial a translated phone number:

```

using Android.Content;
using Android.Telephony;
using Phoneword.Droid;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.Droid
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            var context = MainActivity.Instance;
            if (context == null)
                return false;

            var intent = new Intent (Intent.ActionDial);
            intent.SetData (Uri.Parse ("tel:" + number));

            if (IsIntentAvailable (context, intent)) {
                context.StartActivity (intent);
                return true;
            }

            return false;
        }

        public static bool IsIntentAvailable(Context context, Intent intent)
        {
            var packageManager = context.PackageManager;

            var list = packageManager.QueryIntentServices (intent, 0)
                .Union (packageManager.QueryIntentActivities (intent, 0));

            if (list.Any ())
                return true;

            var manager = TelephonyManager.FromContext (context);
            return manager.PhoneType != PhoneType.None;
        }
    }
}

```

Note that this code assumes that you are using the latest Android API. Save the changes to **PhoneDialer.cs** by pressing **CTRL+S**, and close the file.

21. In **Solution Explorer**, in the **Phoneword.Android** project, double-click **MainActivity.cs** to open it, remove all of the template code and replace it with the following code:

```

using Android.App;
using Android.Content.PM;
using Android.OS;

namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword", Icon = "@mipmap/icon", Theme = "@style/MainTheme", MainLauncher =
true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

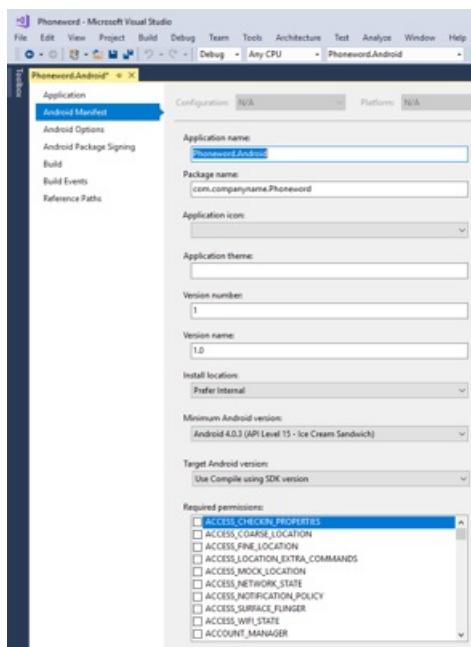
        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}

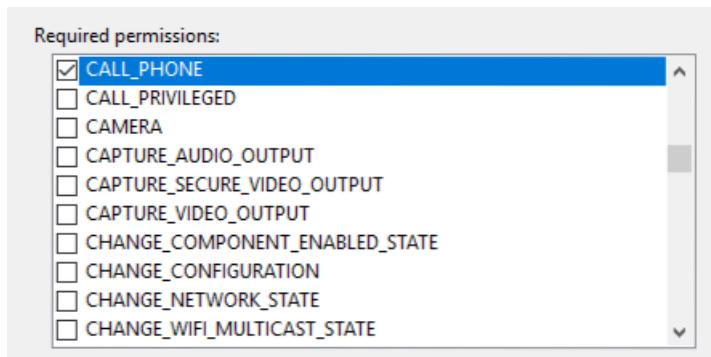
```

Save the changes to **MainActivity.cs** by pressing **CTRL+S**, and close the file.

- In **Solution Explorer**, in the **Phoneword.Android** project, double-click **Properties** and select the **Android Manifest** tab:



- In the **Required permissions** section, enable the **CALL\_PHONE** permission. This gives the application permission to place a phone call:



Save the changes to the manifest by pressing **CTRL+S**, and close the file.

24. Right-click on the Android application project and choose **Set as startup project**.
25. Run the Android app using the "green arrow" toolbar button, or select **Debug > Start Debugging** from the menu.

#### WARNING

Phone calls are not supported on all the simulators, so that feature may not work.

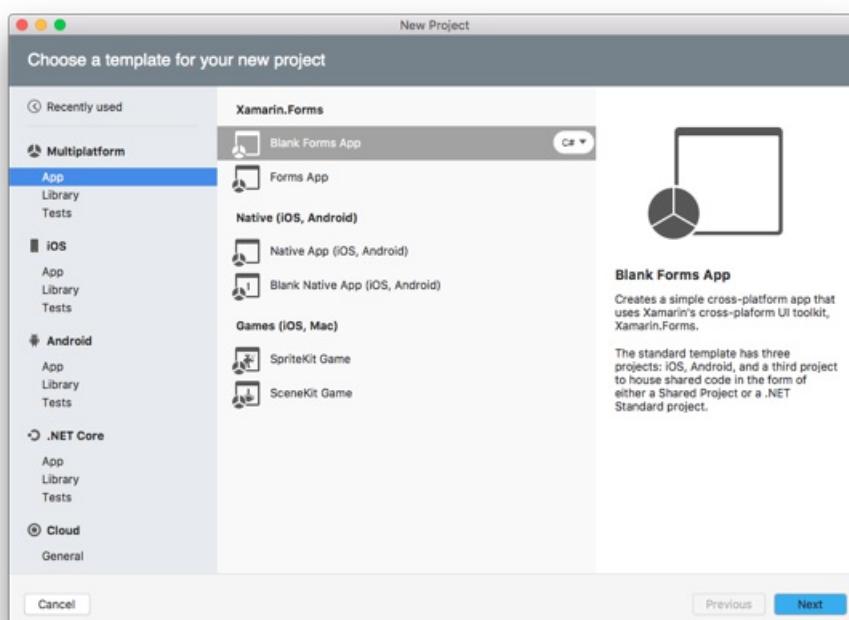
26. If you have an iOS device and meet the Mac system requirements for Xamarin.Forms development, use a similar technique to deploy the app to the iOS device. Alternatively, deploy the app to the [iOS remote simulator](#).

## Get started with Visual Studio for Mac

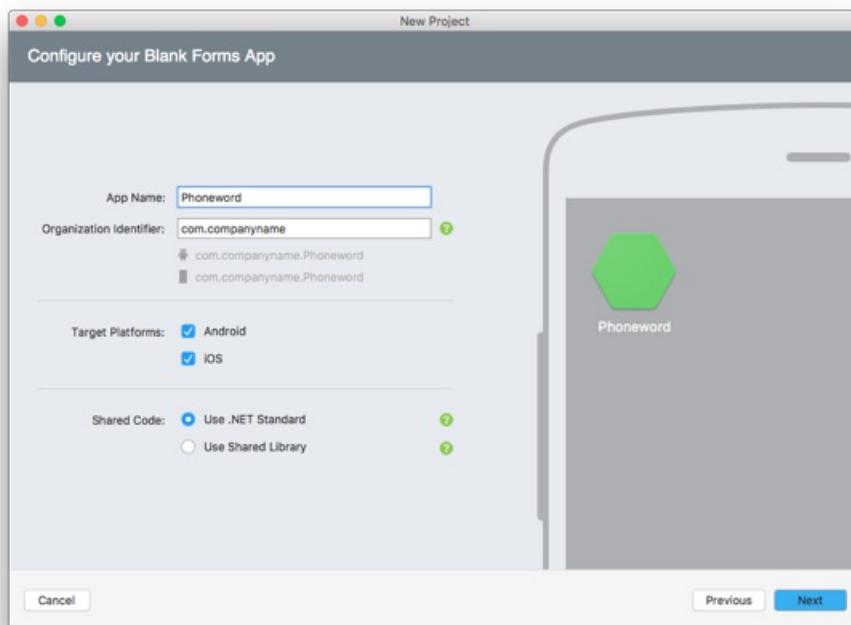
1. Launch Visual Studio for Mac, and on the start page click **New Project...** to create a new project:



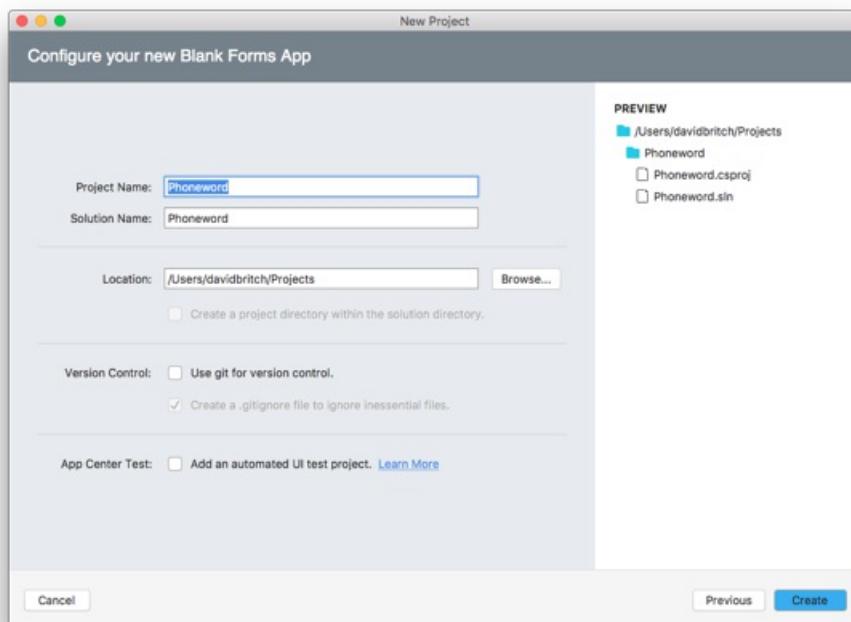
2. In the **Choose a template for your new project** dialog, click **Multiplatform > App**, select the **Blank Forms App** template, and click the **Next** button:



3. In the **Configure your Blank Forms app** dialog, name the new app **Phoneword**, ensure that the **Use .NET Standard** radio button is selected, and click the **Next** button:



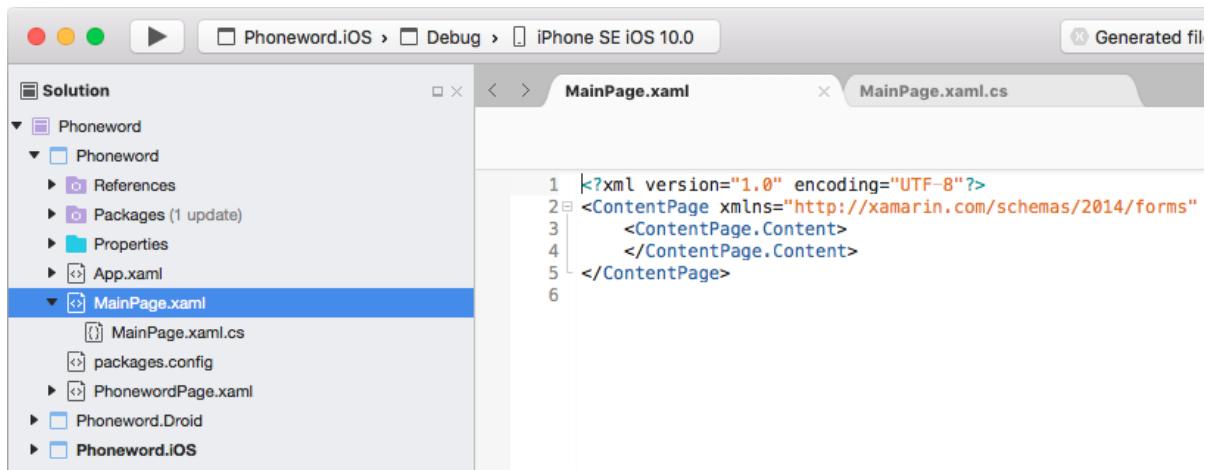
4. In the **Configure your new Blank Forms app** dialog, leave the Solution and Project names set to **Phoneword**, choose a suitable location for the project, and click the **Create** button to create the project:



**NOTE**

The C# and XAML snippets in this quickstart require the solution be named **Phoneword**. Using a different solution name will result in numerous build errors when you copy code from these instructions into the projects.

5. In the **Solution Pad**, double-click **MainPage.xaml** to open it:



6. In **MainPage.xaml**, remove all of the template code and replace it with the following code. This code declaratively defines the user interface for the page:

```
<?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="Phoneword.MainPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
        <Label Text="Enter a Phoneword:" />
        <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
        <Button Text="Translate" Clicked="OnTranslate" />
        <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>
```

Save the changes to **MainPage.xaml** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

7. In the **Solution Pad**, double-click **MainPage.xaml.cs** to open it:



8. In **MainPage.xaml.cs**, remove all of the template code and replace it with the following code. The `OnTranslate` and `OnCall` methods will be executed in response to the **Translate** and **Call** buttons being clicked on the user interface, respectively:

```

using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        string translatedNumber;

        public MainPage ()
        {
            InitializeComponent ();
        }

        void OnTranslate (object sender, EventArgs e)
        {
            translatedNumber = PhonewordTranslator.ToNumber (phoneNumberText.Text);
            if (!string.IsNullOrWhiteSpace (translatedNumber)) {
                callButton.IsEnabled = true;
                callButton.Text = "Call " + translatedNumber;
            } else {
                callButton.IsEnabled = false;
                callButton.Text = "Call";
            }
        }

        async void OnCall (object sender, EventArgs e)
        {
            if (await this.DisplayAlert (
                "Dial a Number",
                "Would you like to call " + translatedNumber + "?",
                "Yes",
                "No")) {
                var dialer = DependencyService.Get<IDialer> ();
                if (dialer != null)
                    dialer.Dial (translatedNumber);
            }
        }
    }
}

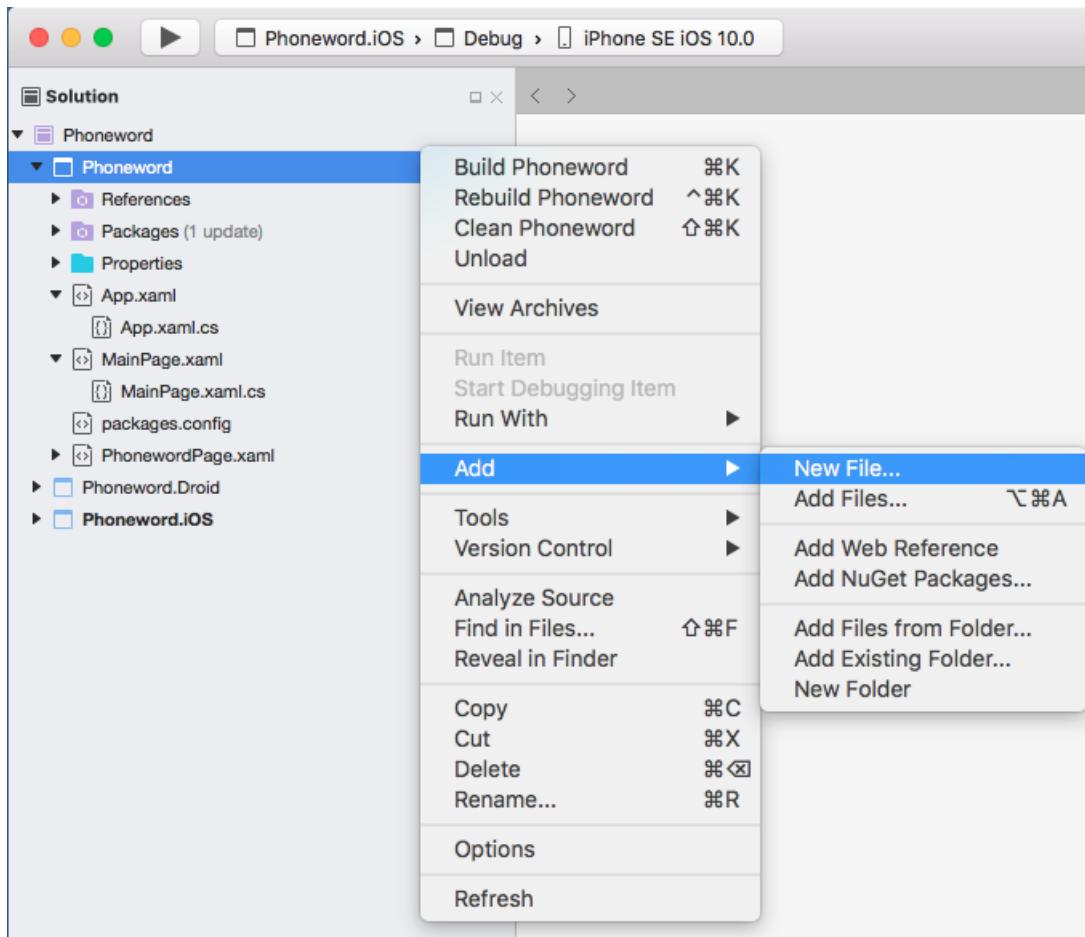
```

#### NOTE

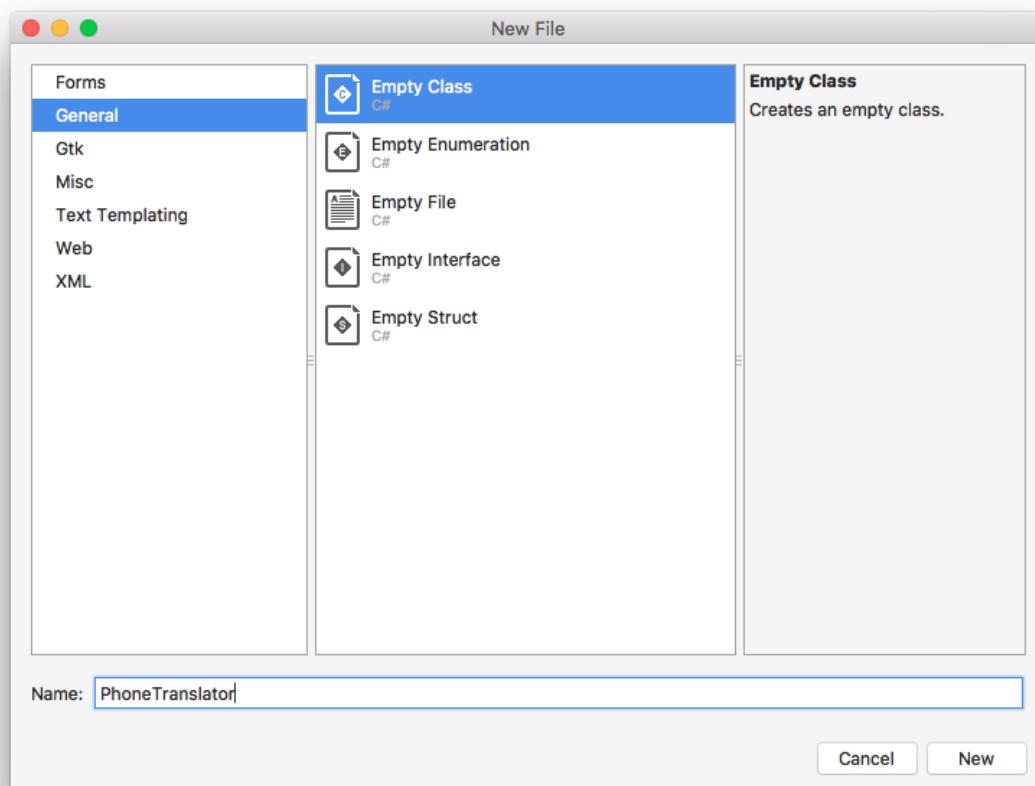
Attempting to build the application at this point will result in errors that will be fixed later.

Save the changes to **MainPage.xaml.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

9. In the **Solution Pad**, select the **Phoneword** project, right-click and select **Add > New File...**



10. In the **New File** dialog, select **General > Empty Class**, name the new file **PhoneTranslator**, and click the **New** button:



11. In **PhoneTranslator.cs**, remove all of the template code and replace it with the following code. This code

will translate a phone word to a phone number:

```
using System.Text;

namespace Phoneword
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return null;

            raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                    newNumber.Append(c);
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                    // Bad character?
                    else
                        return null;
                }
            }
            return newNumber.ToString();
        }

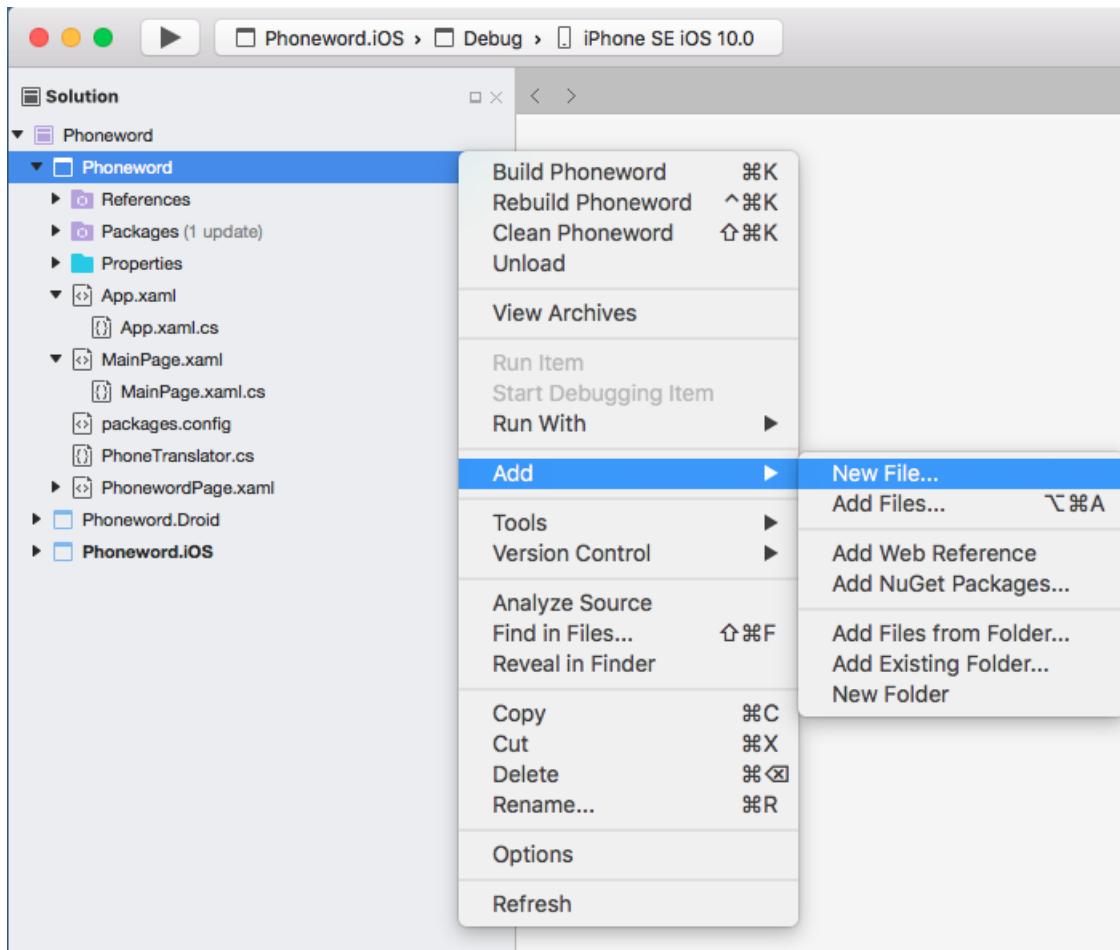
        static bool Contains(this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }

        static readonly string[] digits = {
            "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ"
        };

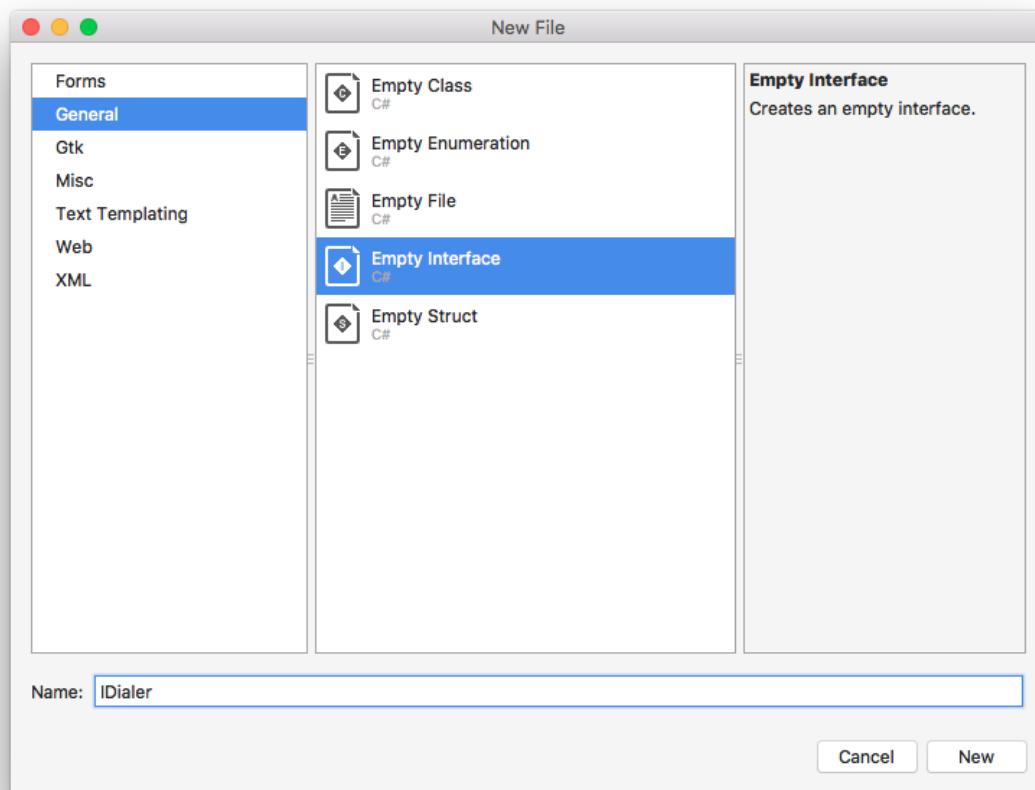
        static int? TranslateToNumber(char c)
        {
            for (int i = 0; i < digits.Length; i++)
            {
                if (digits[i].Contains(c))
                    return 2 + i;
            }
            return null;
        }
    }
}
```

Save the changes to **PhoneTranslator.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

12. In the **Solution Pad**, select the **Phoneword** project, right-click and select **Add > New File...**:



13. In the **New File** dialog, select **General > Empty Interface**, name the new file **IDialer**, and click the **New** button:



14. In **IDialer.cs**, remove all of the template code and replace it with the following code. This code defines a **Dial** method that must be implemented on each platform to dial a translated phone number:

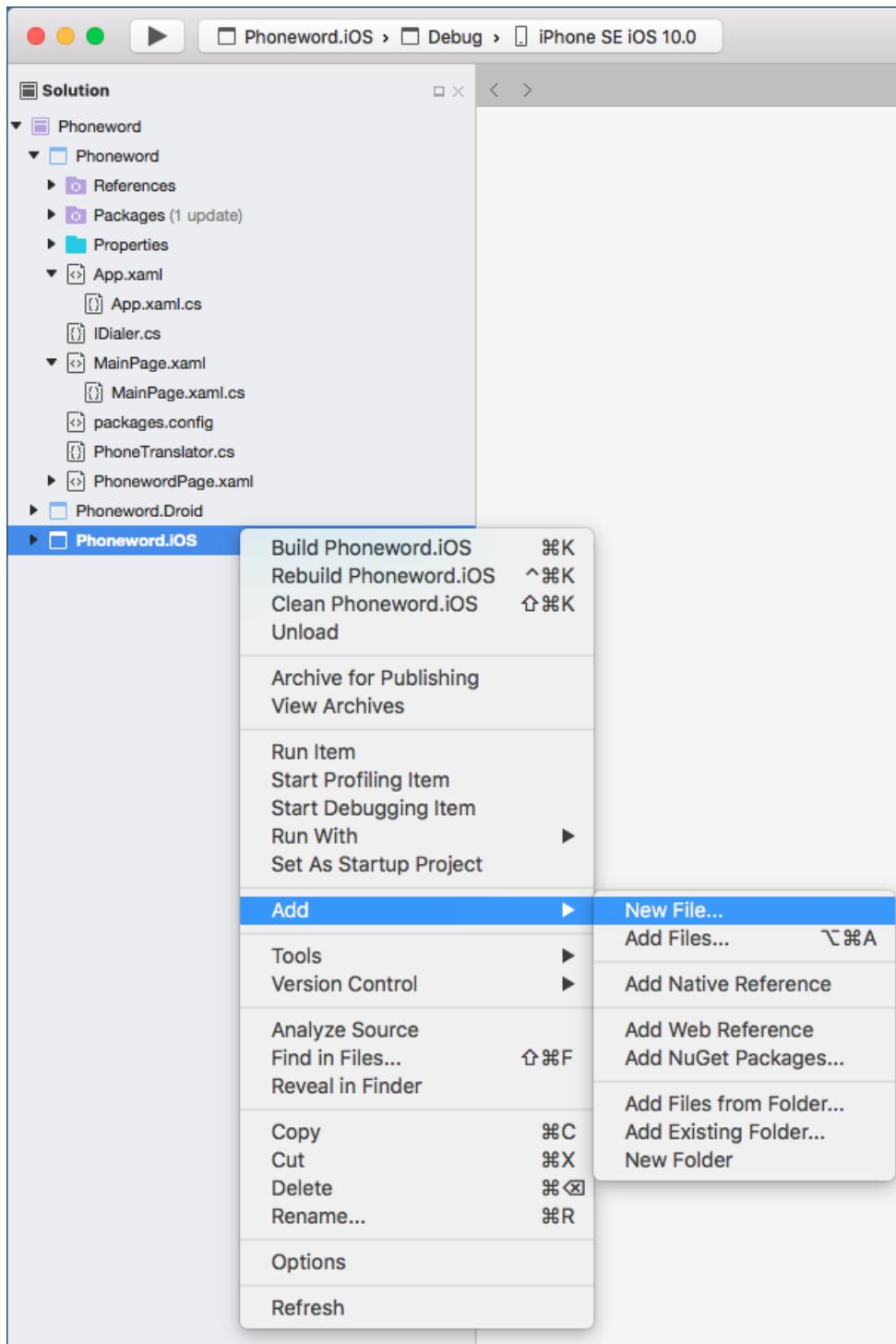
```
namespace Phoneword
{
    public interface IDialer
    {
        bool Dial(string number);
    }
}
```

Save the changes to **IDialer.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

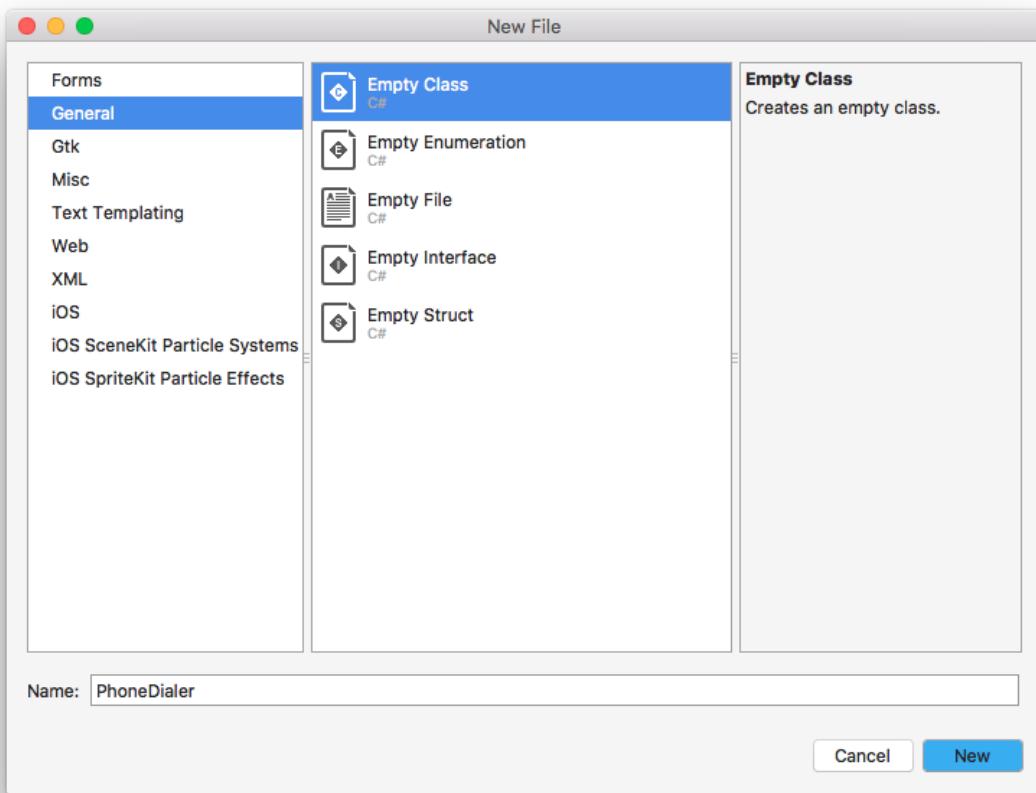
**NOTE**

The common code for the application is now complete. Platform-specific phone dialer code will now be implemented as a [DependencyService](#).

15. In the **Solution Pad**, select the **Phoneword.iOS** project, right-click and select **Add > New File...**:



16. In the **New File** dialog, select **General > Empty Class**, name the new file **PhoneDialer**, and click the **New** button:



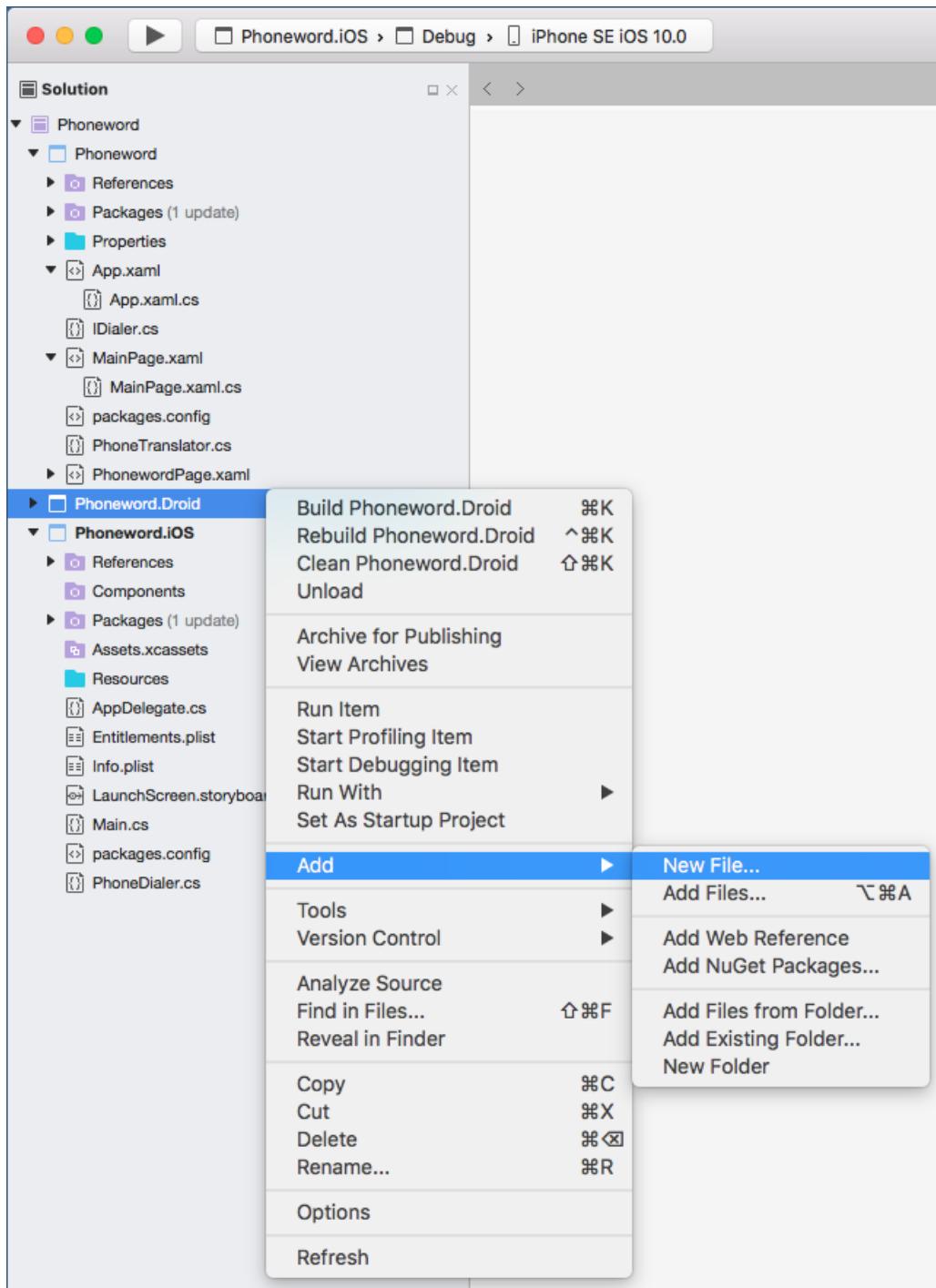
17. In **PhoneDialer.cs**, remove all of the template code and replace it with the following code. This code creates the `Dial` method that will be used on the iOS platform to dial a translated phone number:

```
using Foundation;
using Phoneword.iOS;
using UIKit;
using Xamarin.Forms;

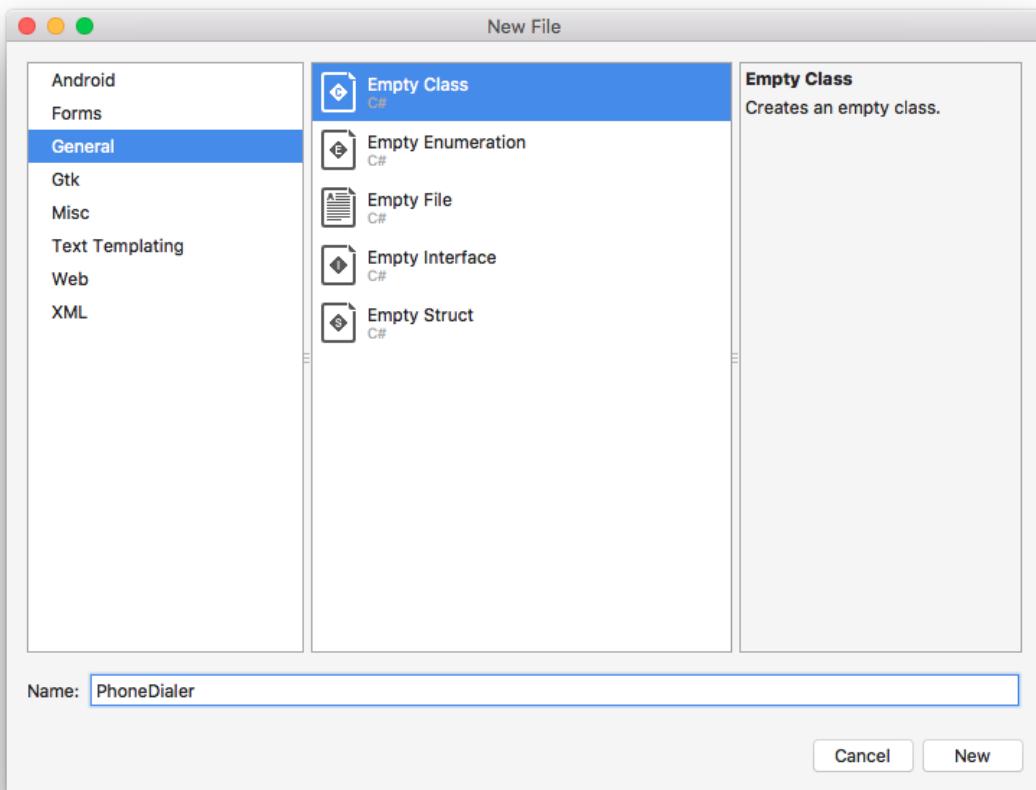
[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.iOS
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            return UIApplication.SharedApplication.OpenUrl (
                new NSUrl ("tel:" + number));
        }
    }
}
```

Save the changes to **PhoneDialer.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

18. In the **Solution Pad**, select the **Phoneword.Droid** project, right-click and select **Add > New File...**:



19. In the **New File** dialog, select **General > Empty Class**, name the new file **PhoneDialer**, and click the **New** button:



20. In **PhoneDialer.cs**, remove all of the template code and replace it with the following code. This code creates the `Dial` method that will be used on the Android platform to dial a translated phone number:

```

using Android.Content;
using Android.Telephony;
using Phoneword.Droid;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.Droid
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            var context = MainActivity.Instance;
            if (context == null)
                return false;

            var intent = new Intent (Intent.ActionDial);
            intent.SetData (Uri.Parse ("tel:" + number));

            if (IsIntentAvailable (context, intent)) {
                context.StartActivity (intent);
                return true;
            }

            return false;
        }

        public static bool IsIntentAvailable(Context context, Intent intent)
        {
            var packageManager = context.PackageManager;

            var list = packageManager.QueryIntentServices (intent, 0)
                .Union (packageManager.QueryIntentActivities (intent, 0));

            if (list.Any ())
                return true;

            var manager = TelephonyManager.FromContext (context);
            return manager.PhoneType != PhoneType.None;
        }
    }
}

```

Note that this code assumes that you are using the latest Android API. Save the changes to **PhoneDialer.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

21. In the **Solution Pad**, in the **Phoneword.Droid** project, double click **MainActivity.cs** to open it, remove all of the template code and replace it with the following code:

```

using Android.App;
using Android.Content.PM;
using Android.OS;

namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword", Icon = "@mipmap/icon", Theme = "@style/MainTheme", MainLauncher =
true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

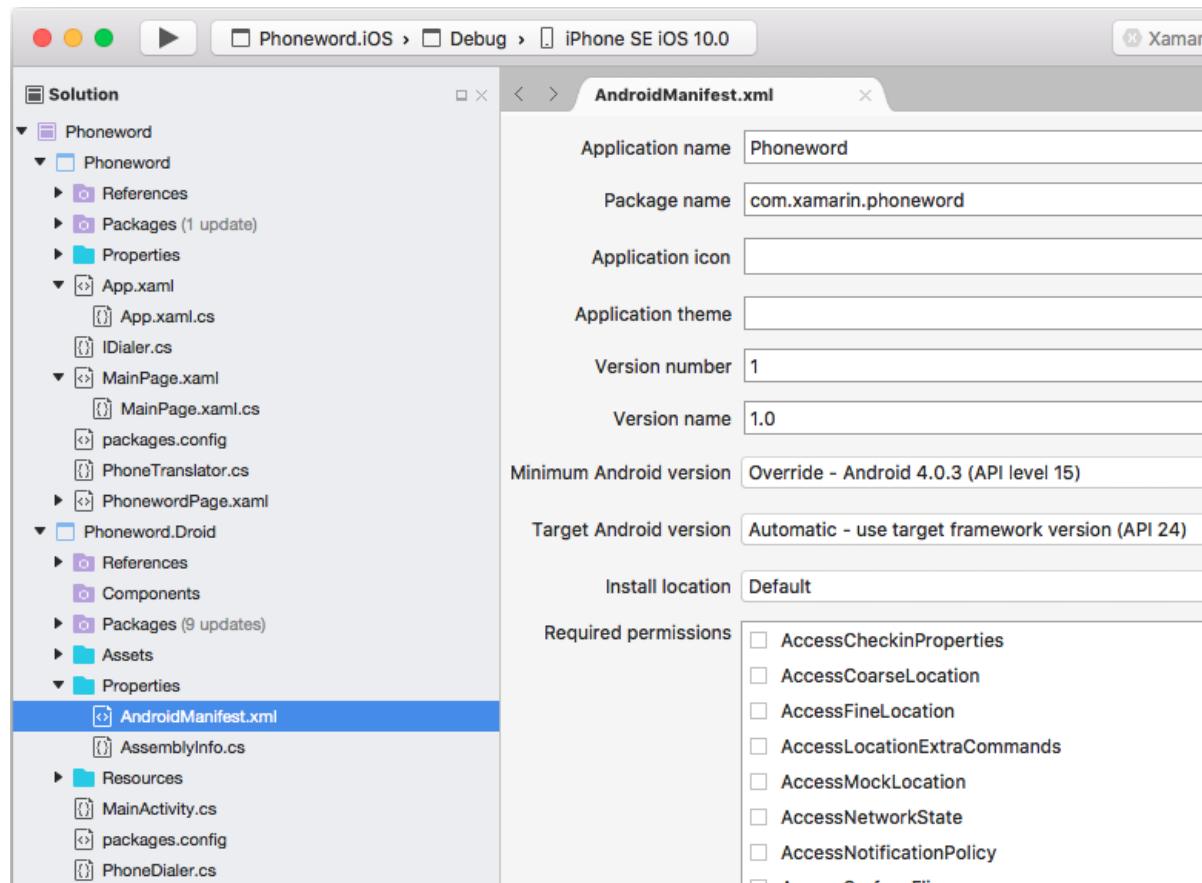
        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}

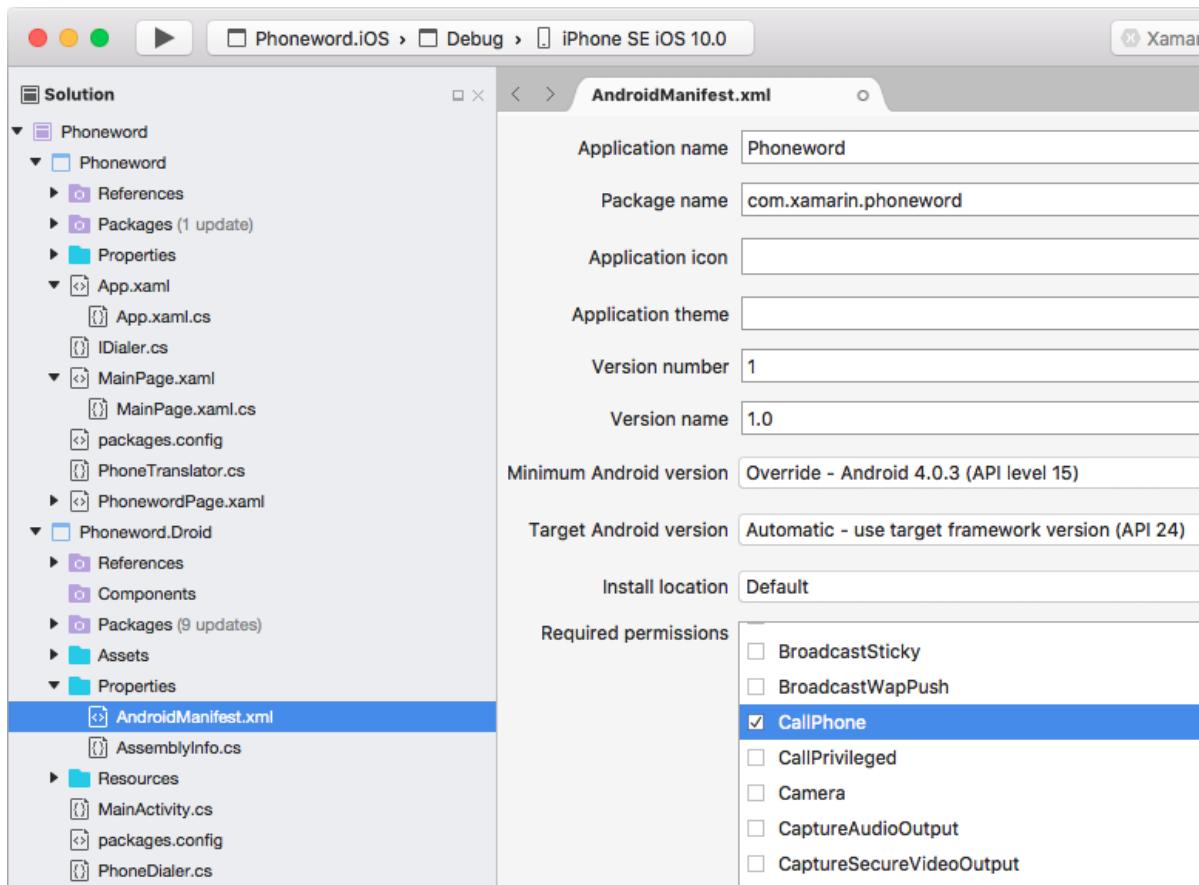
```

Save the changes to **MainActivity.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

22. In the **Solution Pad**, expand the **Properties** folder and double-click the **AndroidManifest.xml** file:

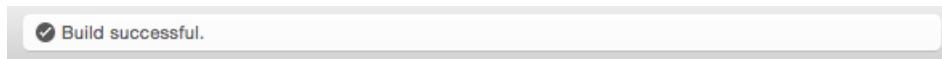


23. In the **Required permissions** section, enable the **CallPhone** permission. This gives the application permission to place a phone call:



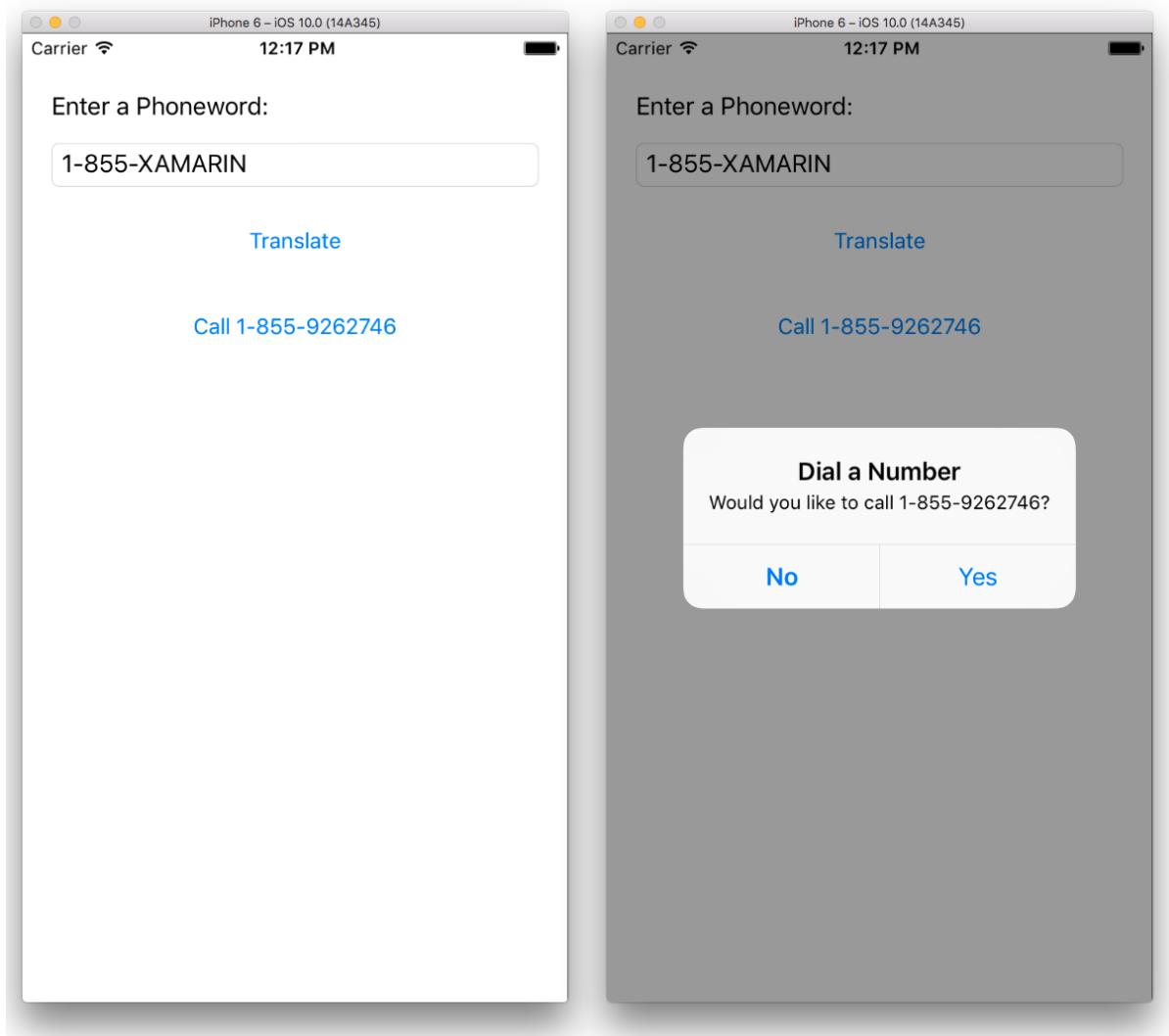
Save the changes to **AndroidManifest.xml** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

24. In Visual Studio for Mac, select the **Build > Build All** menu item (or press **⌘ + B**). The application will build and a success message will appear in the Visual Studio for Mac toolbar.



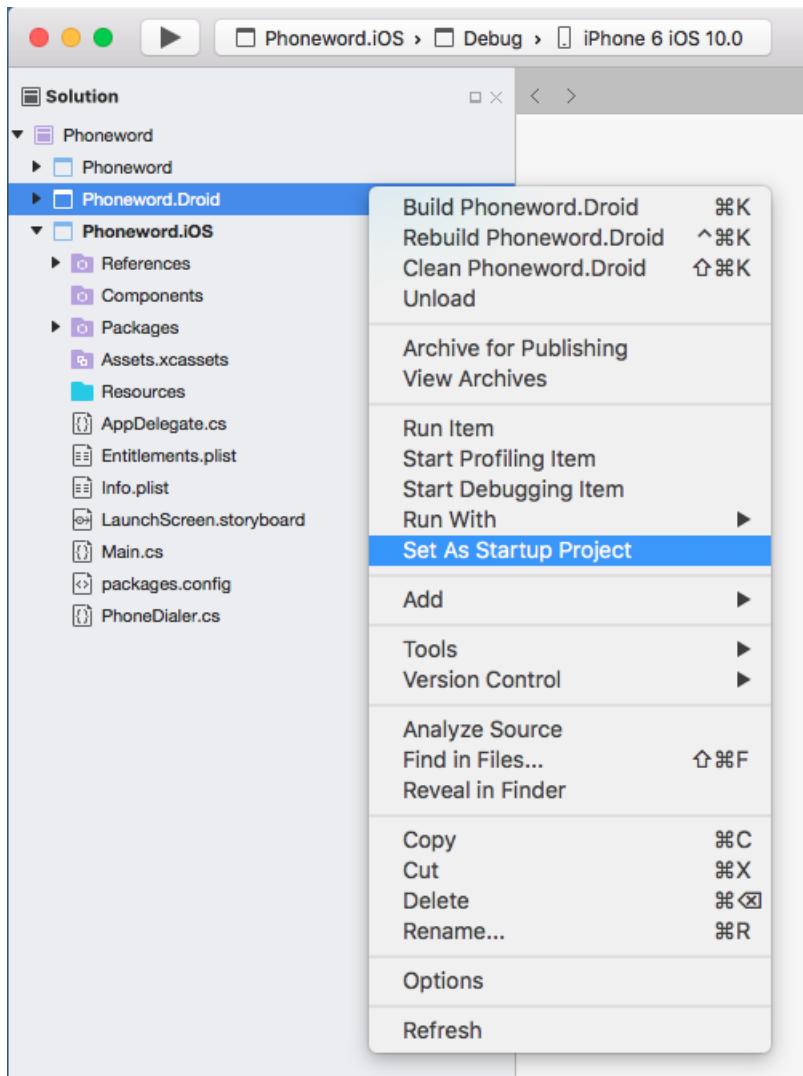
25. If there are errors, repeat the previous steps and correct any mistakes until the application builds successfully.
26. In the Visual Studio for Mac toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application inside the iOS Simulator:



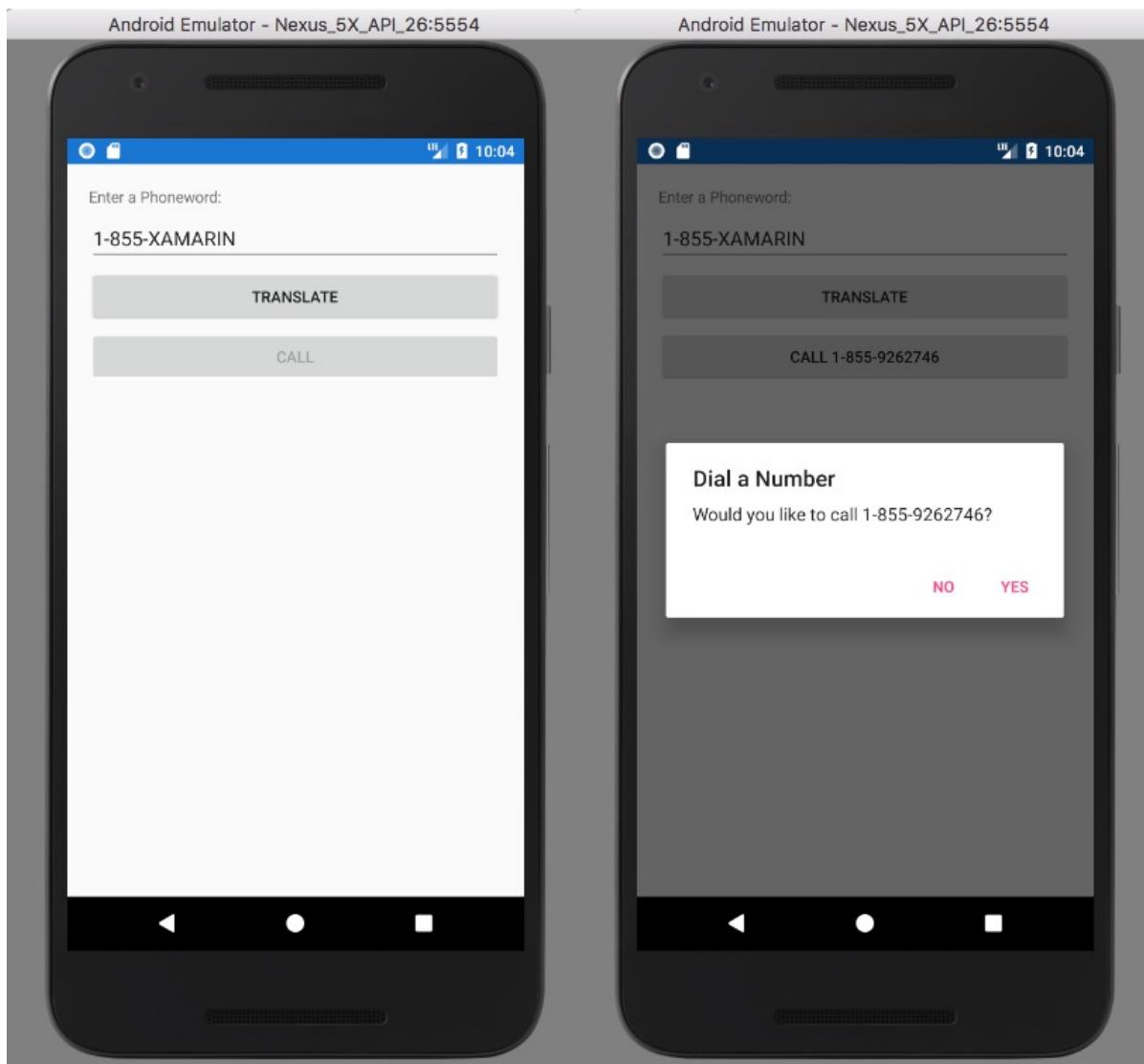


Note: phone calls are not supported in the iOS Simulator.

27. In the **Solution Pad**, select the **Phoneword.Droid** project, right-click and select **Set As Startup Project**:



28. In the Visual Studio for Mac toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application inside an Android emulator:



#### WARNING

Phone calls are not supported in Android emulators.

Congratulations on completing a Xamarin.Forms application. The [next topic](#) in this guide reviews the steps that were taken in this walkthrough to gain an understanding of the fundamentals of application development using Xamarin.Forms.

## Related Links

- [Accessing Native Features via the DependencyService](#)
- [Phoneword \(sample\)](#)

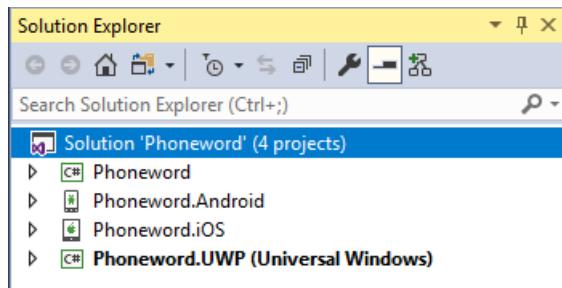
# Xamarin.Forms Deep Dive

10/12/2018 • 10 minutes to read • [Edit Online](#)

In the [Xamarin.Forms Quickstart](#), the Phoneword application was built. This article reviews what has been built to gain an understanding of the fundamentals of how Xamarin.Forms applications work.

## Introduction to Visual Studio

Visual Studio organizes code into *Solutions* and *Projects*. A solution is a container that can hold one or more projects. A project can be an application, a supporting library, a test application, and more. The Phoneword application consists of one solution containing four projects, as shown in the following screenshot:

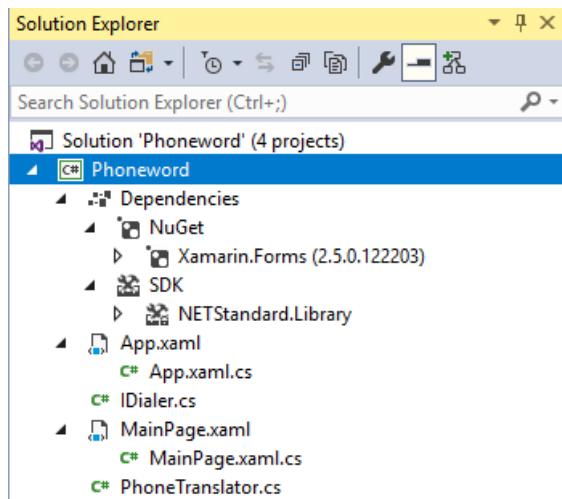


The projects are:

- Phoneword – This project is the .NET Standard library project that holds all of the shared code and shared UI.
- Phoneword.Android – This project holds Android-specific code and is the entry point for the Android application.
- Phoneword.iOS – This project holds iOS specific code and is the entry point for the iOS application.
- Phoneword.UWP – This project holds Universal Windows Platform (UWP) specific code and is the entry point for the UWP application.

## Anatomy of a Xamarin.Forms Application

The following screenshot shows the contents of the Phoneword .NET Standard library project in Visual Studio:



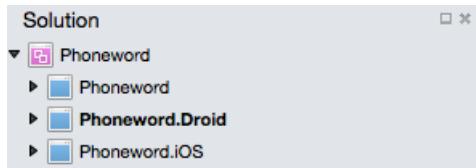
The project has a **Dependencies** node that contains **NuGet** and **SDK** nodes:

- **NuGet** – the Xamarin.Forms NuGet package that has been added to the project.
- **SDK** – the `NETStandard.Library` metapackage that references the complete set of NuGet packages that define

.NET Standard.

## Introduction to Visual Studio for Mac

Visual Studio for Mac follows the Visual Studio practice of organizing code into *Solutions* and *Projects*. A solution is a container that can hold one or more projects. A project can be an application, a supporting library, a test application, and more. The Phoneword application consists of one solution containing three projects, as shown in the following screenshot:

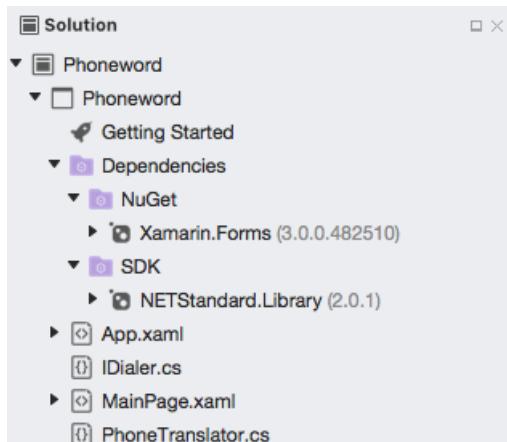


The projects are:

- Phoneword – This project is the .NET Standard library project that holds all of the shared code and shared UI.
- Phoneword.Droid – This project holds Android specific code and is the entry point for Android applications.
- Phoneword.iOS – This project holds iOS specific code and is the entry point for iOS applications.

## Anatomy of a Xamarin.Forms Application

The following screenshot shows the contents of the Phoneword .NET Standard library project in Visual Studio for Mac:



The project has a **Dependencies** node that contains **NuGet** and **SDK** nodes:

- **NuGet** – the Xamarin.Forms NuGet package that has been added to the project.
- **SDK** – the `NETStandard.Library` metapackage that references the complete set of NuGet packages that define .NET Standard.

The project also consists of a number of files:

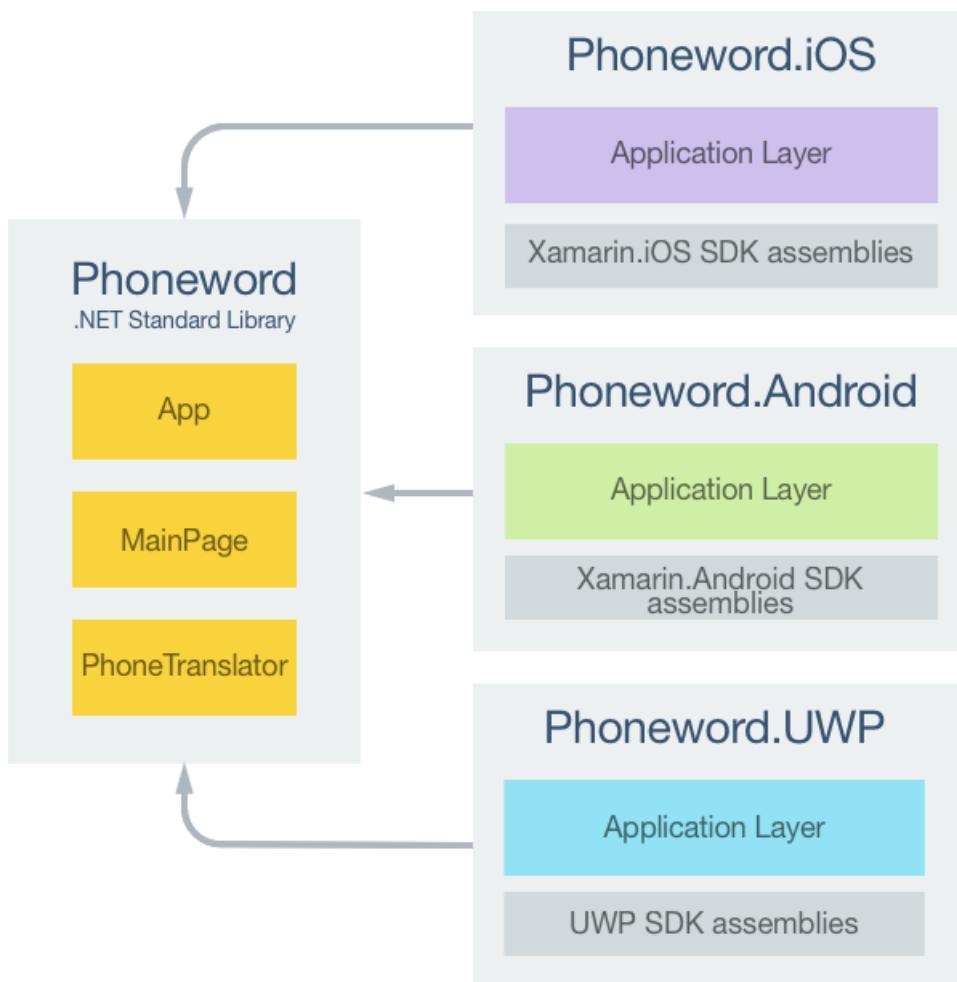
- **App.xaml** – The XAML markup for the `App` class, which defines a resource dictionary for the application.
- **App.xaml.cs** – The code-behind for the `App` class, which is responsible for instantiating the first page that will be displayed by the application on each platform, and for handling application lifecycle events.
- **IDialer.cs** – The `IDialer` interface, which specifies that the `Dial` method must be provided by any implementing classes.
- **MainPage.xaml** – The XAML markup for the `MainPage` class, which defines the UI for the page shown when the application launches.
- **MainPage.xaml.cs** – The code-behind for the `MainPage` class, which contains the business logic that is executed when the user interacts with the page.

- **PhoneTranslator.cs** – The business logic that is responsible for converting a phone word to a phone number, which is invoked from **MainPage.xaml.cs**.

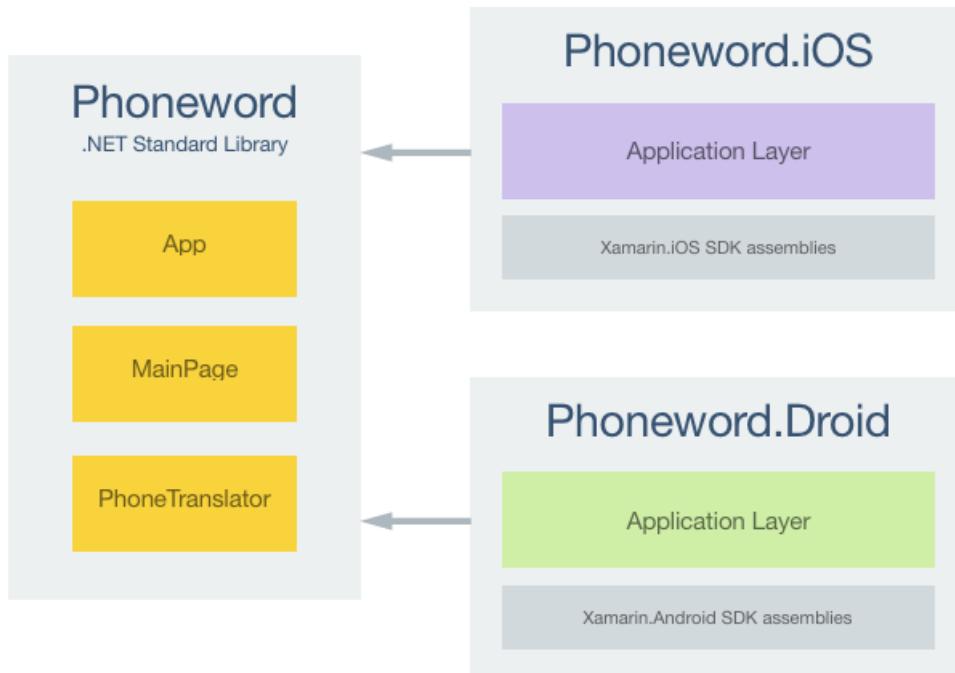
For more information about the anatomy of a Xamarin.iOS application, see [Anatomy of a Xamarin.iOS Application](#). For more information about the anatomy of a Xamarin.Android application, see [Anatomy of a Xamarin.Android Application](#).

## Architecture and Application Fundamentals

A Xamarin.Forms application is architected in the same way as a traditional cross-platform application. Shared code is typically placed in a .NET Standard library, and platform-specific applications consume the shared code. The following diagram shows an overview of this relationship for the Phoneword application:



A Xamarin.Forms application is architected in the same way as a traditional cross-platform application. Shared code is typically placed in a .NET Standard library, and platform-specific applications consume the shared code. The following diagram shows an overview of this relationship for the Phoneword application:



To maximize the reuse of startup code, Xamarin.Forms applications have a single class named `App` that is responsible for instantiating the first page that will be displayed by the application on each platform, as shown in the following code example:

```
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new MainPage();
        }
        ...
    }
}
```

This code sets the `MainPage` property of the `App` class to a new instance of the `MainPage` class. In addition, the `XamlCompilation` attribute turns on the XAML compiler, so that XAML is compiled directly into intermediate language. For more information, see [XAML Compilation](#).

## Launching the Application on Each Platform

### iOS

To launch the initial Xamarin.Forms page in iOS, the Phoneword.iOS project includes the `AppDelegate` class that inherits from the `FormsApplicationDelegate` class, as shown in the following code example:

```

namespace Phoneword.iOS
{
    [Register ("AppDelegate")]
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        public override bool FinishedLaunching (UIApplication app, NSDictionary options)
        {
            global::Xamarin.Forms.Forms.Init ();
            LoadApplication (new App ());
            return base.FinishedLaunching (app, options);
        }
    }
}

```

The `FinishedLaunching` override initializes the Xamarin.Forms framework by calling the `Init` method. This causes the iOS-specific implementation of Xamarin.Forms to be loaded in the application before the root view controller is set by the call to the `LoadApplication` method.

## Android

To launch the initial Xamarin.Forms page in Android, the Phoneword.Droid project includes code that creates an `Activity` with the `MainLauncher` attribute, with the activity inheriting from the `FormsAppCompatActivity` class, as shown in the following code example:

```

namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword",
        Icon = "@mipmap/icon",
        Theme = "@style/MainTheme",
        MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}

```

The `OnCreate` override initializes the Xamarin.Forms framework by calling the `Init` method. This causes the Android-specific implementation of Xamarin.Forms to be loaded in the application before the Xamarin.Forms application is loaded. In addition, the `MainActivity` class stores a reference to itself in the `Instance` property. The `Instance` property is known as the local context, and is referenced from the `PhoneDialer` class.

## Universal Windows Platform

In Universal Windows Platform (UWP) applications, the `Init` method that initializes the Xamarin.Forms framework is invoked from the `App` class:

```
Xamarin.Forms.Forms.Init (e);

if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    ...
}
```

This causes the UWP-specific implementation of Xamarin.Forms to be loaded in the application. The initial Xamarin.Forms page is launched by the `MainPage` class, as demonstrated in the following code example:

```
namespace Phoneword.UWP
{
    public sealed partial class MainPage
    {
        public MainPage()
        {
            this.InitializeComponent();
            this.LoadApplication(new Phoneword.App());
        }
    }
}
```

The Xamarin.Forms application is loaded with the `LoadApplication` method.

#### NOTE

Universal Windows Platform (UWP) apps can be built with Xamarin.Forms, but only using Visual Studio on Windows.

## User Interface

There are four main control groups used to create the user interface of a Xamarin.Forms application.

1. **Pages** – Xamarin.Forms pages represent cross-platform mobile application screens. The Phoneword application uses the `ContentPage` class to display a single screen. For more information about pages, see [Xamarin.Forms Pages](#).
2. **Layouts** – Xamarin.Forms layouts are containers used to compose views into logical structures. The Phoneword application uses the `StackLayout` class to arrange controls in a vertical stack. For more information about layouts, see [Xamarin.Forms Layouts](#).
3. **Views** – Xamarin.Forms views are the controls displayed on the user interface, such as labels, buttons, and text entry boxes. The Phoneword application uses the `Label`, `Entry`, and `Button` controls. For more information about views, see [Xamarin.Forms Views](#).
4. **Cells** – Xamarin.Forms cells are specialized elements used for items in a list, and describe how each item in a list should be drawn. The Phoneword application does not make use of any cells. For more information about cells, see [Xamarin.Forms Cells](#).

At runtime, each control will be mapped to its native equivalent, which is what will be rendered.

When the Phoneword application is run on any platform, it displays a single screen that corresponds to a `Page` in Xamarin.Forms. A `Page` represents a *ViewGroup* in Android, a *View Controller* in iOS, or a *Page* on the Universal Windows Platform. The Phoneword application also instantiates a `ContentPage` object that represents the `MainPage` class, whose XAML markup is shown in the following code example:

```

<?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="Phoneword.MainPage">
    ...
    <StackLayout>
        <Label Text="Enter a Phoneword:" />
        <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
        <Button x:Name="translateButton" Text="Translate" Clicked="OnTranslate" />
        <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>

```

The `MainPage` class uses a `StackLayout` control to automatically arrange controls on the screen regardless of the screen size. Each child element is positioned one after the other, vertically in the order in which they are added. The `StackLayout` control contains a `Label` control to display text on the page, an `Entry` control to accept textual user input, and two `Button` controls used to execute code in response to touch events.

For more information about XAML in Xamarin.Forms, see [Xamarin.Forms XAML Basics](#).

## Responding to User Interaction

An object defined in XAML can fire an event that is handled in the code-behind file. The following code example shows the `OnTranslate` method in the code-behind for the `MainPage` class, which is executed in response to the `Clicked` event firing on the *Translate* button.

```

void OnTranslate(object sender, EventArgs e)
{
    translatedNumber = Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (!string.IsNullOrWhiteSpace(translatedNumber)) {
        callButton.IsEnabled = true;
        callButton.Text = "Call " + translatedNumber;
    } else {
        callButton.IsEnabled = false;
        callButton.Text = "Call";
    }
}

```

The `OnTranslate` method translates the phoneword into its corresponding phone number, and in response, sets properties on the call button. The code-behind file for a XAML class can access an object defined in XAML using the name assigned to it with the `x:Name` attribute. The value assigned to this attribute has the same rules as C# variables, in that it must begin with a letter or underscore and contain no embedded spaces.

The wiring of the translate button to the `OnTranslate` method occurs in the XAML markup for the `MainPage` class:

```
<Button x:Name="translateButton" Text="Translate" Clicked="OnTranslate" />
```

## Additional Concepts Introduced in Phoneword

The Phoneword application for Xamarin.Forms has introduced several concepts not covered in this article. These concepts include:

- Enabling and disabling buttons. A `Button` can be toggled on or off by changing its `IsEnabled` property. For example, the following code example disables the `callButton`:

```
callButton.IsEnabled = false;
```

- Displaying an alert dialog. When the user presses the call **Button** the Phoneword application shows an *Alert Dialog* with the option to place or cancel a call. The `DisplayAlert` method is used to create the dialog, as shown in the following code example:

```
await this.DisplayAlert (
    "Dial a Number",
    "Would you like to call " + translatedNumber + "?",
    "Yes",
    "No");
```

- Accessing native features via the `DependencyService` class. The Phoneword application uses the `DependencyService` class to resolve the `IDialer` interface to platform-specific phone dialing implementations, as shown in the following code example from the Phoneword project:

```
async void OnCall (object sender, EventArgs e)
{
    ...
    var dialer = DependencyService.Get<IDialer> ();
    ...
}
```

For more information about the `DependencyService` class, see [Accessing Native Features via the DependencyService](#).

- Placing a phone call with a URL. The Phoneword application uses `OpenURL` to launch the system phone app. The URL consists of a `tel:` prefix followed by the phone number to be called, as shown in the following code example from the iOS project:

```
return UIApplication.SharedApplication.OpenUrl (new NSUrl ("tel:" + number));
```

- Tweaking the platform layout. The `Device` class enables developers to customize the application layout and functionality on a per-platform basis, as shown in the following code example that uses different `Padding` values on different platforms to correctly display each page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" ... >
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>
```

For more information about platform tweaks, see [Device Class](#).

## Testing and Deployment

Visual Studio for Mac and Visual Studio both provide many options for testing and deploying an application. Debugging applications is a common part of the application development lifecycle and helps to diagnose code issues. For more information, see [Set a Breakpoint, Step Through Code](#), and [Output Information to the Log Window](#).

Simulators are a good place to start deploying and testing an application, and feature useful functionality for testing applications. However, users will not consume the final application in a simulator, so applications should be

tested on real devices early and often. For more information about iOS device provisioning, see [Device Provisioning](#). For more information about Android device provisioning, see [Set Up Device for Development](#).

## Summary

This article has examined the fundamentals of application development using Xamarin.Forms. Topics covered included the anatomy of a Xamarin.Forms application, architecture and application fundamentals, and the user interface.

In the next section of this guide the application will be extended to include multiple screens, to explore more advanced Xamarin.Forms architecture and concepts.

# Hello, Xamarin.Forms Multiscreen

7/25/2018 • 2 minutes to read • [Edit Online](#)

*This guide expands the Phoneword application created in the Hello, Xamarin.Forms guide to navigate to a second screen. Topics covered include page navigation and data binding to a collection.*

## Part 1: Quickstart

The first part of this guide demonstrates how to add a second screen to the Phoneword application to keep track of the call history for the application.

## Part 2: Deep Dive

The second part of this guide reviews what has been built, to develop an understanding of page navigation and data binding in a Xamarin.Forms application.

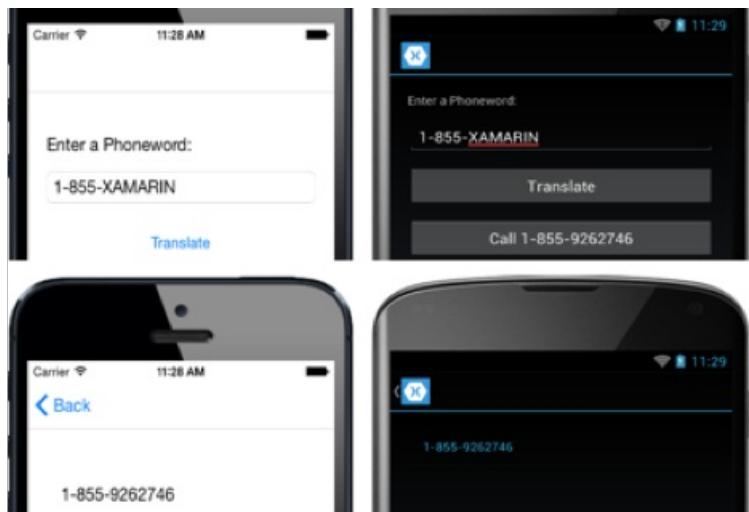
## Related Links

- [Introduction to Xamarin.Forms](#)
- [Debugging in Visual Studio](#)
- [Visual Studio for Mac Recipes - Debugging](#)
- [Free Self-Guided Learning \(video\)](#)
- [Getting Started with Xamarin \(video\)](#)

# Xamarin.Forms Multiscreen Quickstart

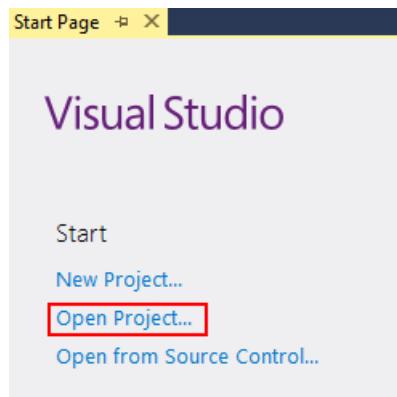
9/13/2018 • 6 minutes to read • [Edit Online](#)

This quickstart demonstrates how to extend the Phoneword application by adding a second screen to keep track of the call history for the application. The final application is shown below:

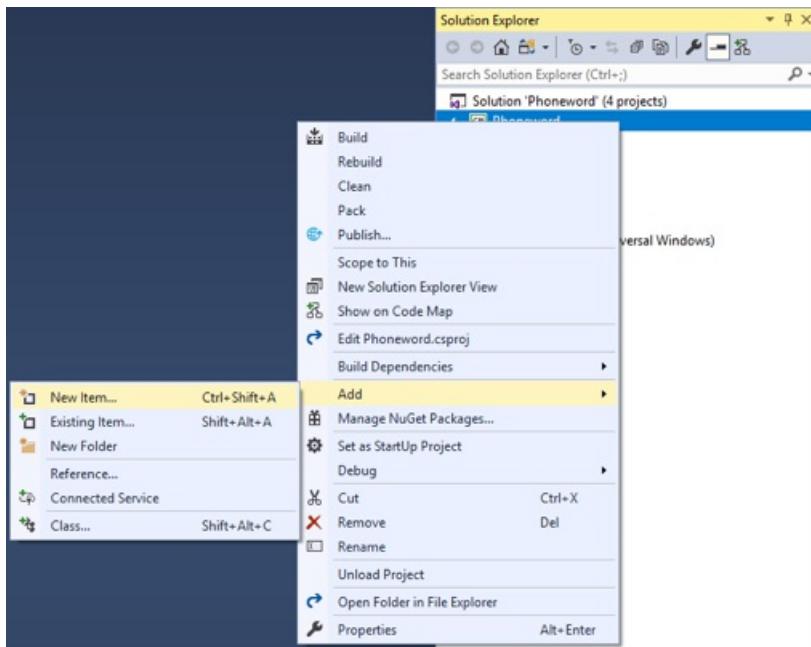


## Update the app with Visual Studio

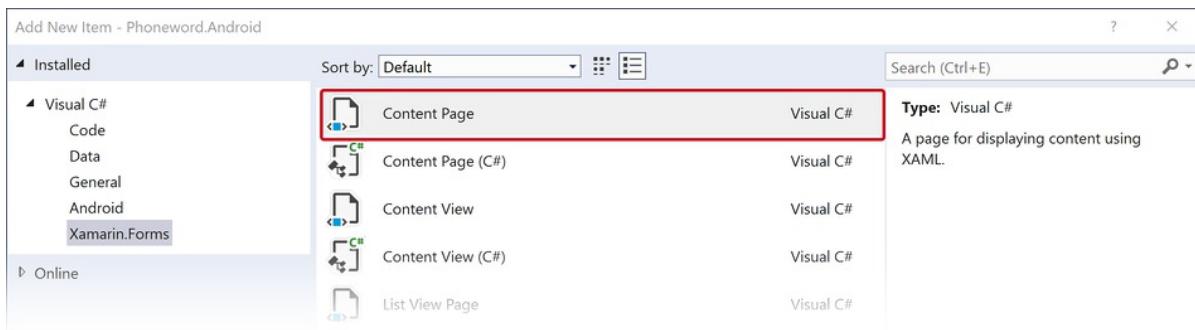
1. Launch Visual Studio. On the start page click **Open Project...**, and in the **Open Project** dialog select the solution file for the Phoneword project:



2. In **Solution Explorer**, right click on the **Phoneword** project and select **Add > New Item...**:



- In the **Add New Item** dialog, select **Visual C# Items > Xamarin.Forms > Content Page**, name the new item **CallHistoryPage**, and click the **Add** button. This will add a page named **CallHistoryPage** to the project:

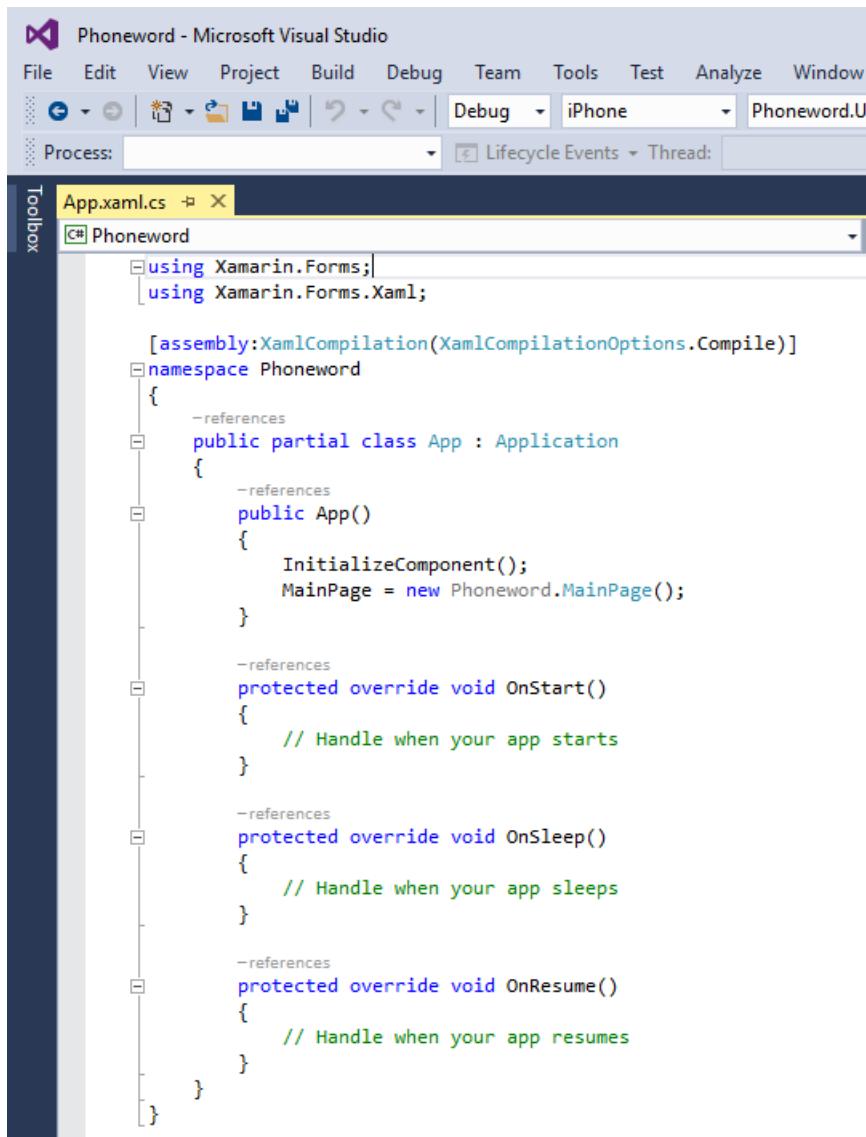


- In **CallHistoryPage.xaml**, remove all of the template code and replace it with the following code. This code declaratively defines the user interface for the page:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Phoneword.CallHistoryPage"
    Title="Call History">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
        <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
    </StackLayout>
</ContentPage>
```

Save the changes to **CallHistoryPage.xaml** by pressing **CTRL+S**, and close the file.

- In **Solution Explorer**, double-click the **App.xaml.cs** file in the shared **Phoneword** project to open it:



6. In **App.xaml.cs**, import the `System.Collections.Generic` namespace, add the declaration of the `PhoneNumbers` property, initialize the property in the `App` constructor, and initialize the `MainPage` property to be a `NavigationPage`. The `PhoneNumbers` collection will be used to store a list of each translated phone number called by the application:

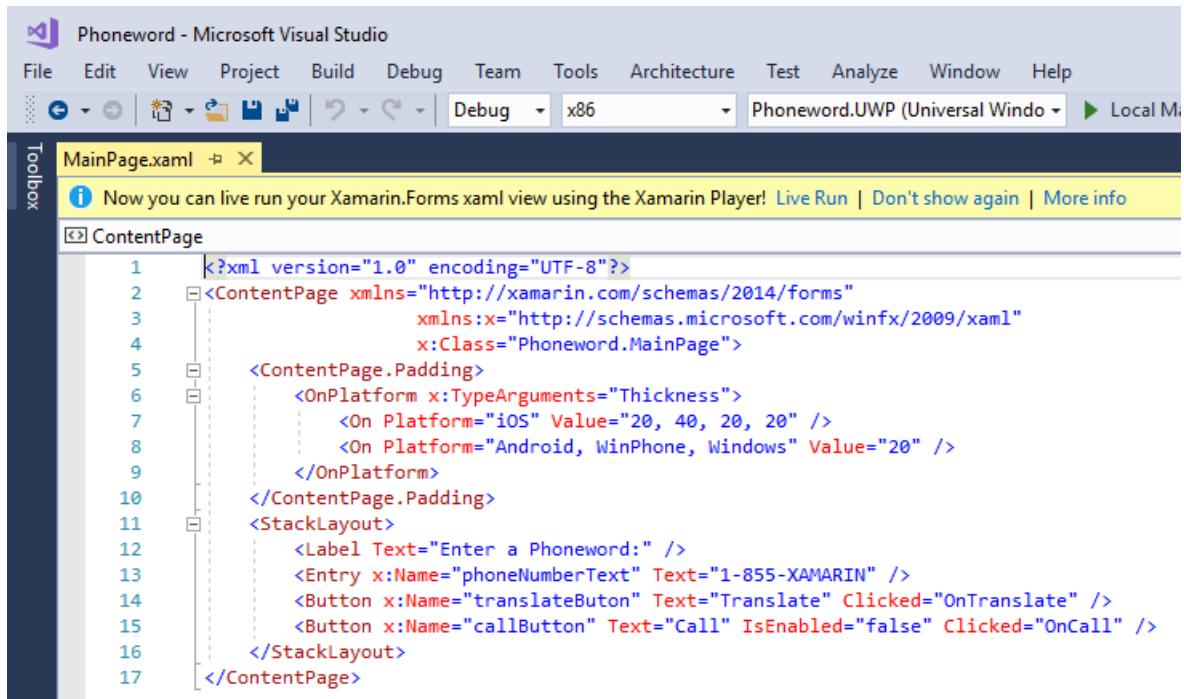
```
using System.Collections.Generic;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public static IList<string> PhoneNumbers { get; set; }

        public App()
        {
            InitializeComponent();
            PhoneNumbers = new List<string>();
            MainPage = new NavigationPage(new MainPage());
        }
        ...
    }
}
```

Save the changes to **App.xaml.cs** by pressing **CTRL+S**, and close the file.

7. In **Solution Explorer**, double-click the **MainPage.xaml** file in the shared **Phoneword** project to open it:



The screenshot shows the Microsoft Visual Studio interface with the title bar "Phoneword - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Architecture, Test, Analyze, Window, and Help. The toolbar has standard icons like Open, Save, and Build. The status bar shows "Debug x86 Phoneword.UWP (Universal Windo Local Ma". The main window displays the XAML code for "MainPage.xaml". The code defines a ContentPage with a StackLayout containing a Label, an Entry, and two Buttons. A message at the top says "Now you can live run your Xamarin.Forms xaml view using the Xamarin Player! Live Run | Don't show again | More info".

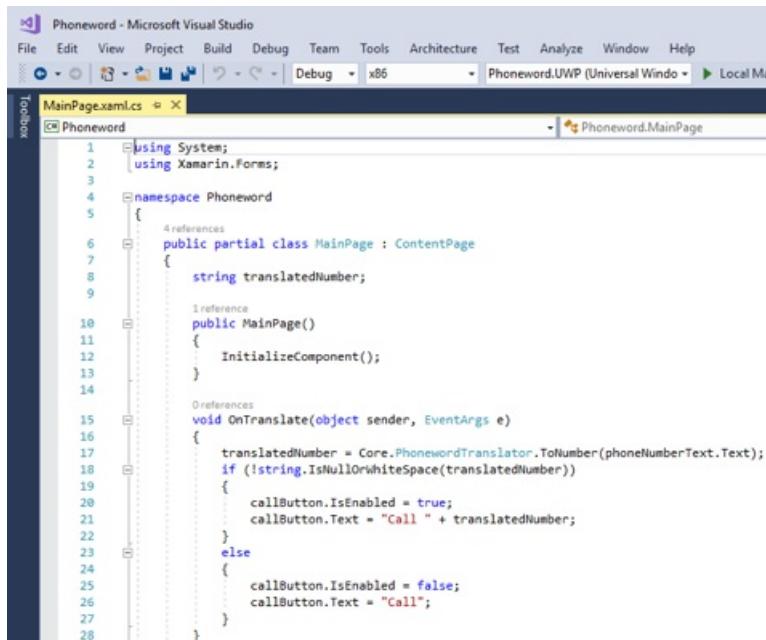
```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Phoneword.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, WinPhone, Windows" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
        <Label Text="Enter a Phoneword:" />
        <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
        <Button x:Name="translateButton" Text="Translate" Clicked="OnTranslate" />
        <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>
```

8. In **MainPage.xaml**, add a **Button** control at the end of the **StackLayout** control. The button will be used to navigate to the call history page:

```
<StackLayout VerticalOptions="FillAndExpand"
    HorizontalOptions="FillAndExpand"
    Orientation="Vertical"
    Spacing="15">
    ...
    <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    <Button x:Name="callHistoryButton" Text="Call History" IsEnabled="false"
        Clicked="OnCallHistory" />
</StackLayout>
```

Save the changes to **MainPage.xaml** by pressing **CTRL+S**, and close the file.

9. In **Solution Explorer**, double-click **MainPage.xaml.cs** to open it:



The screenshot shows the Microsoft Visual Studio interface with the title bar "Phoneword - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Architecture, Test, Analyze, Window, and Help. The toolbar has standard icons like Open, Save, and Build. The status bar shows "Debug x86 Phoneword.UWP (Universal Windo Local Ma". The main window displays the C# code for "MainPage.xaml.cs". The code defines a MainPage class that implements the ContentPage interface. It contains properties for translatedNumber and phoneNumberText, and methods for OnTranslate and OnCall. The OnTranslate method converts the phone number to a translated string and updates the call button's text and enabled state. The OnCall method is currently empty.

```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        string translatedNumber;

        public MainPage()
        {
            InitializeComponent();
        }

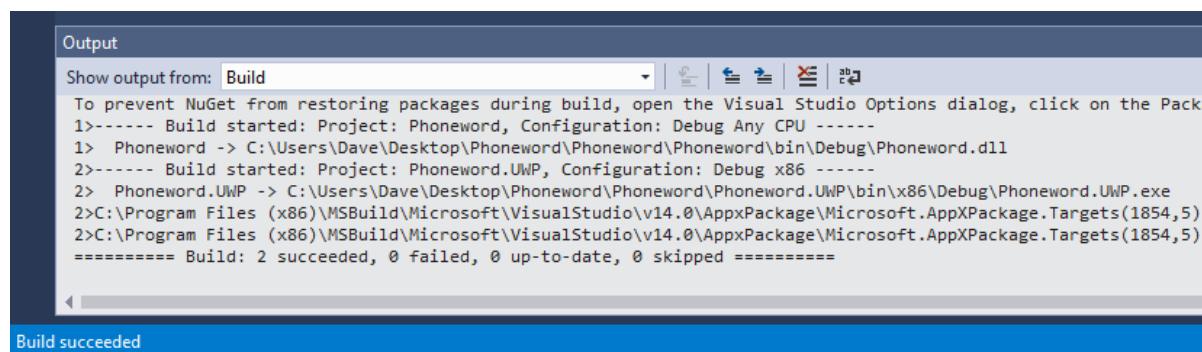
        void OnTranslate(object sender, EventArgs e)
        {
            translatedNumber = Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);
            if (!string.IsNullOrWhiteSpace(translatedNumber))
            {
                callButton.IsEnabled = true;
                callButton.Text = "Call " + translatedNumber;
            }
            else
            {
                callButton.IsEnabled = false;
                callButton.Text = "Call";
            }
        }
    }
}
```

10. In **MainPage.xaml.cs**, add the **OnCallHistory** event handler method, and modify the **OnCall** event

handler method to add the translated phone number to the `App.PhoneNumbers` collection and enable the `callHistoryButton`, provided that the `dialer` variable is not `null`:

Save the changes to **MainPage.xaml.cs** by pressing **CTRL+S**, and close the file.

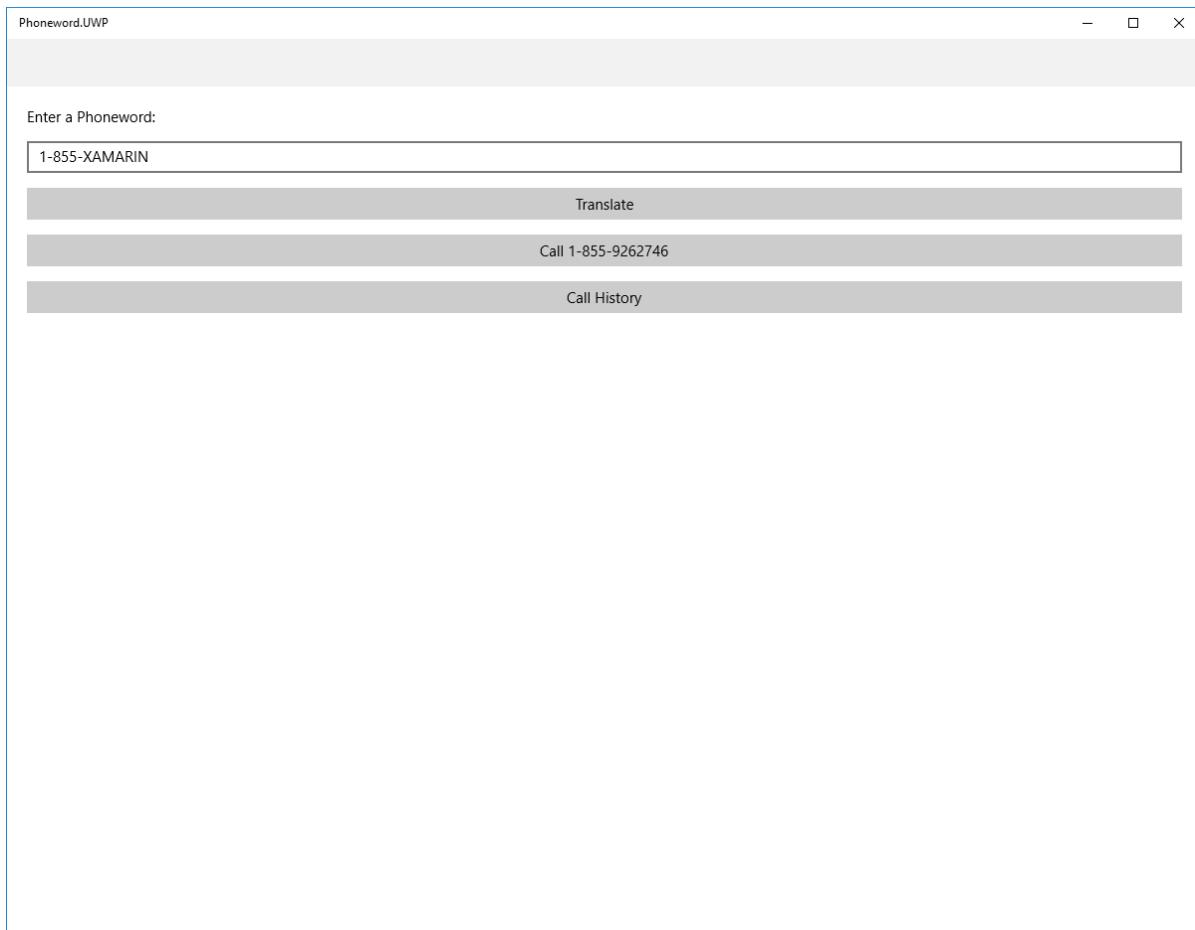
11. In Visual Studio, select the **Build > Build Solution** menu item (or press **CTRL+SHIFT+B**). The application will build and a success message will appear in the Visual Studio status bar:



If there are errors, repeat the previous steps and correct any mistakes until the application builds successfully.

12. In the Visual Studio toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application:





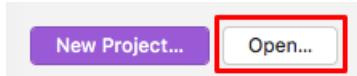
13. In **Solution Explorer**, right click on the **Phoneword.Droid** project and select **Set as StartUp Project**.
14. In the Visual Studio toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application inside an Android emulator.
15. If you have an iOS device and meet the Mac system requirements for Xamarin.Forms development, use a similar technique to deploy the app to the iOS device. Alternatively, deploy the app to the [iOS remote simulator](#).

**NOTE**

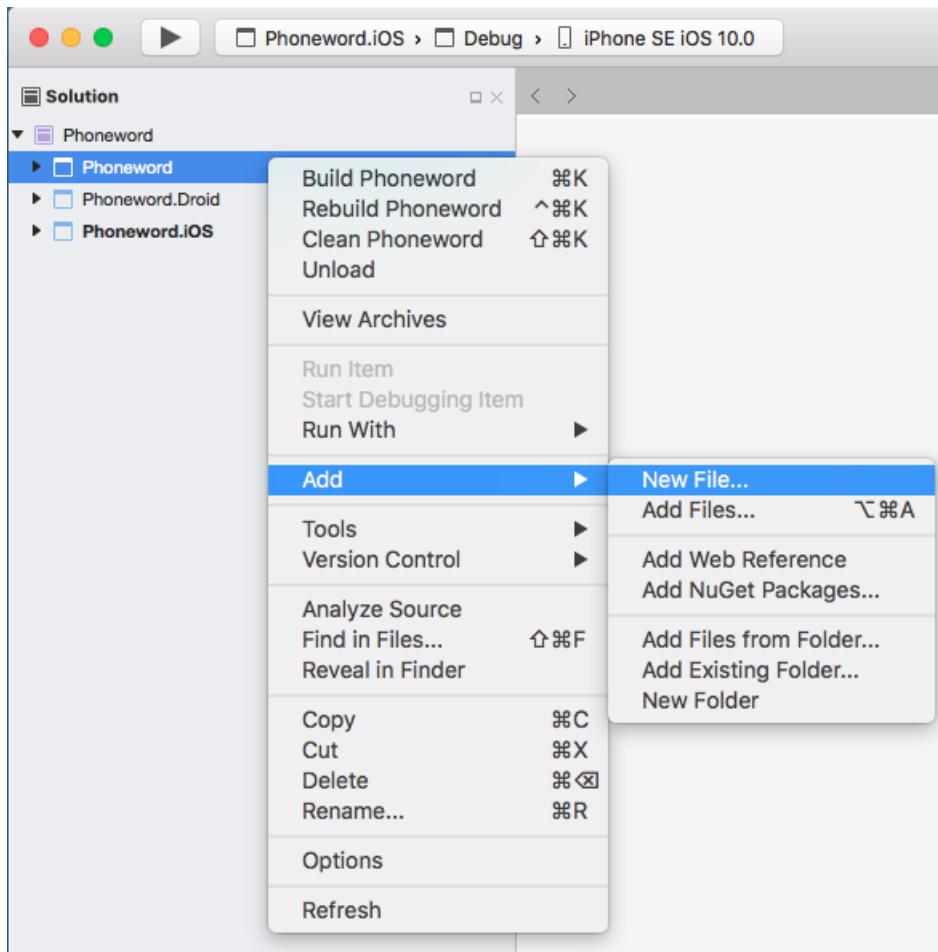
Phone calls are not supported in device emulators.

## Update the app with Visual Studio for Mac

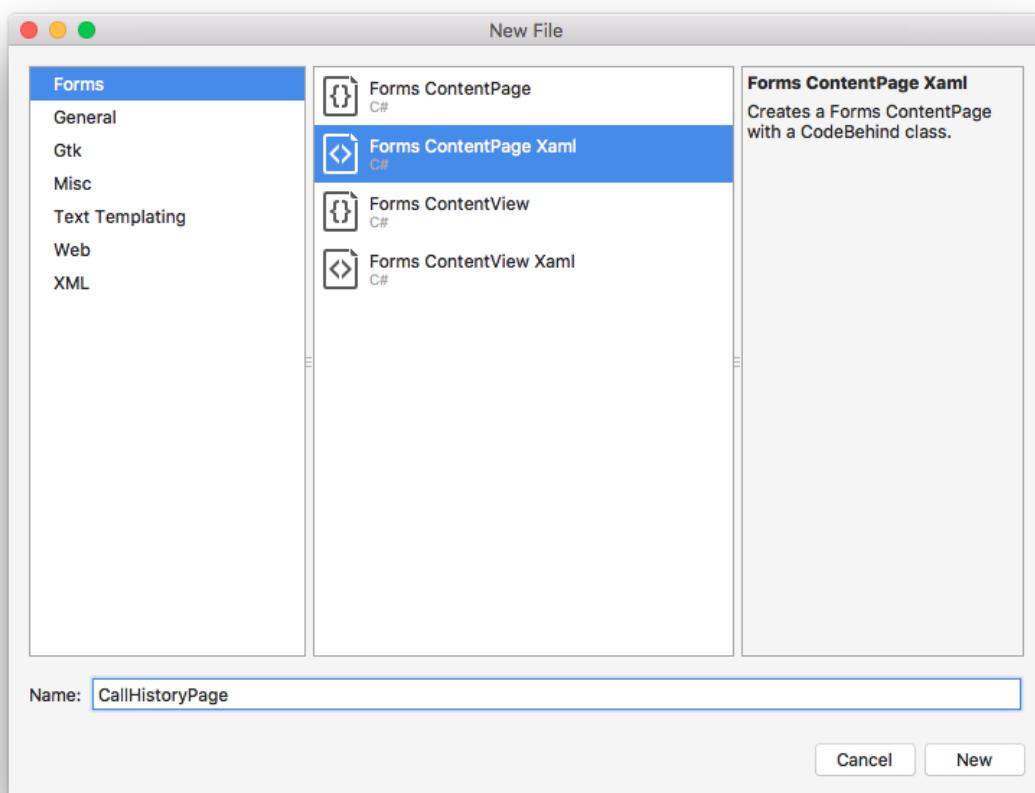
1. Launch Visual Studio for Mac. On the start page click **Open...**, and in the dialog select the solution file for the Phoneword project:



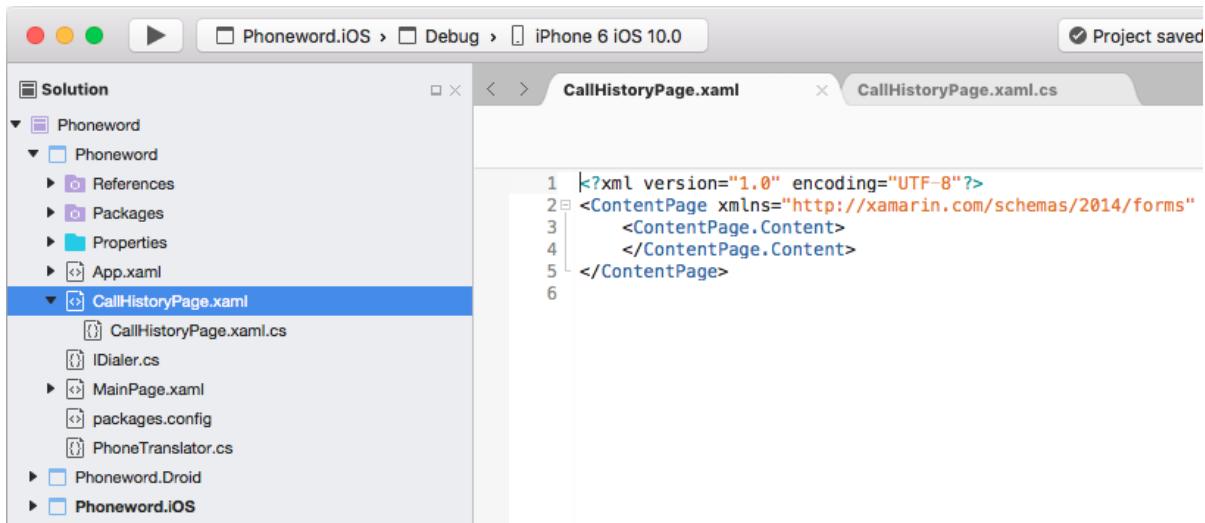
2. In the **Solution Pad**, select the **Phoneword** project, right-click and select **Add > New File...**:



3. In the **New File** dialog, select **Forms > Forms ContentPage Xaml**, name the new file **CallHistoryPage**, and click the **New** button. This will add a page named **CallHistoryPage** to the project:



4. In the **Solution Pad**, double-click **CallHistoryPage.xaml** to open it:

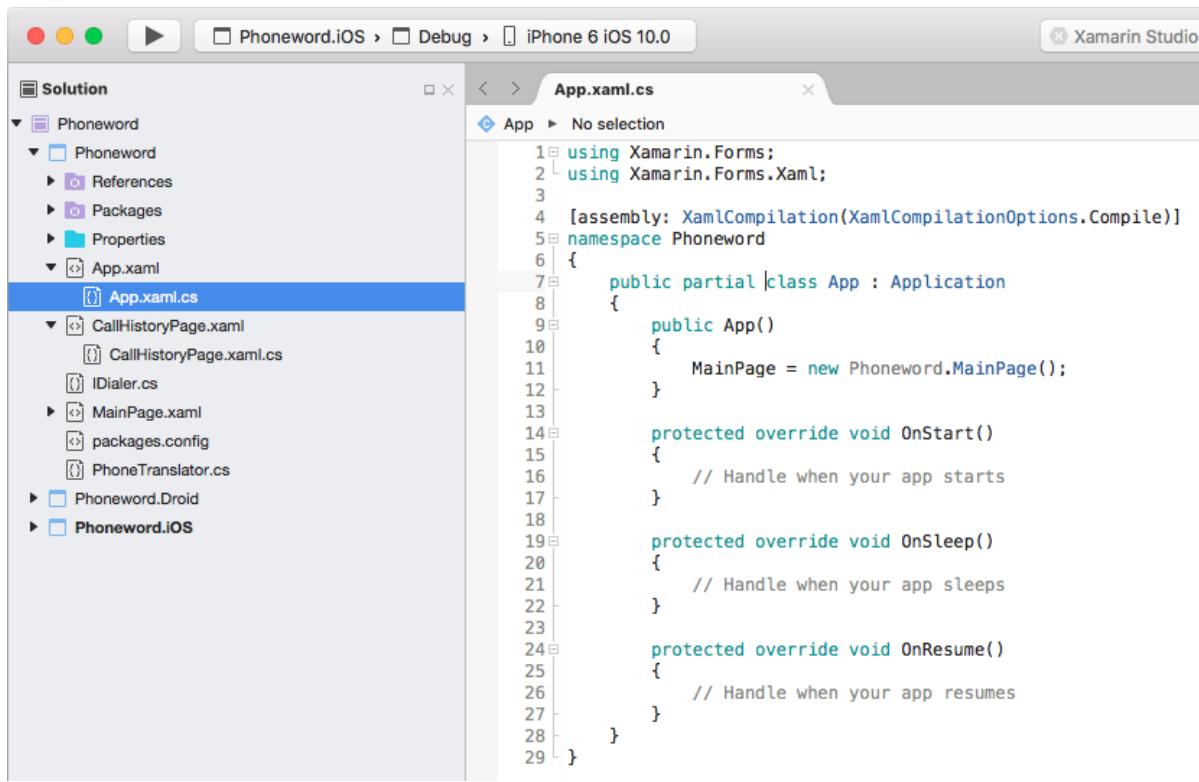


5. In **CallHistoryPage.xaml**, remove all of the template code and replace it with the following code. This code declaratively defines the user interface for the page:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Phoneword.CallHistoryPage"
    Title="Call History">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
        <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
    </StackLayout>
</ContentPage>
```

Save the changes to **CallHistoryPage.xaml** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

6. In the **Solution Pad**, double-click **App.xaml.cs** to open it:



7. In **App.xaml.cs**, import the `System.Collections.Generic` namespace, add the declaration of the `PhoneNumbers` property, initialize the property in the `App` constructor, and initialize the `MainPage` property to be a `NavigationPage`. The `PhoneNumbers` collection will be used to store a list of each translated phone number called by the application:

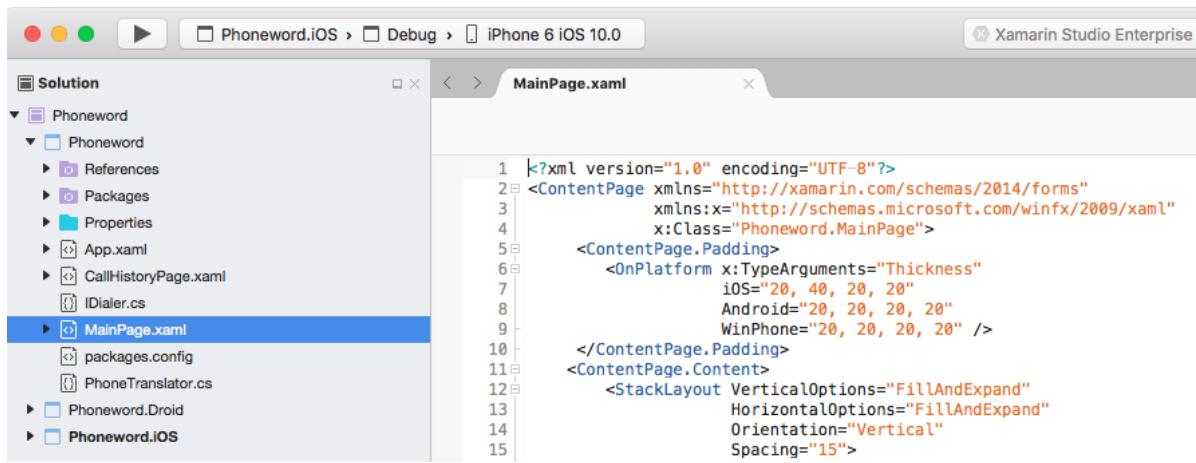
```
using System.Collections.Generic;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public static IList<string> PhoneNumbers { get; set; }

        public App()
        {
            InitializeComponent();
            PhoneNumbers = new List<string>();
            MainPage = new NavigationPage(new MainPage());
        }
        ...
    }
}
```

Save the changes to **App.xaml.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

8. In the **Solution Pad**, double-click **MainPage.xaml** to open it:



9. In **MainPage.xaml**, add a `Button` control at the end of the `StackLayout` control. The button will be used to navigate to the call history page:

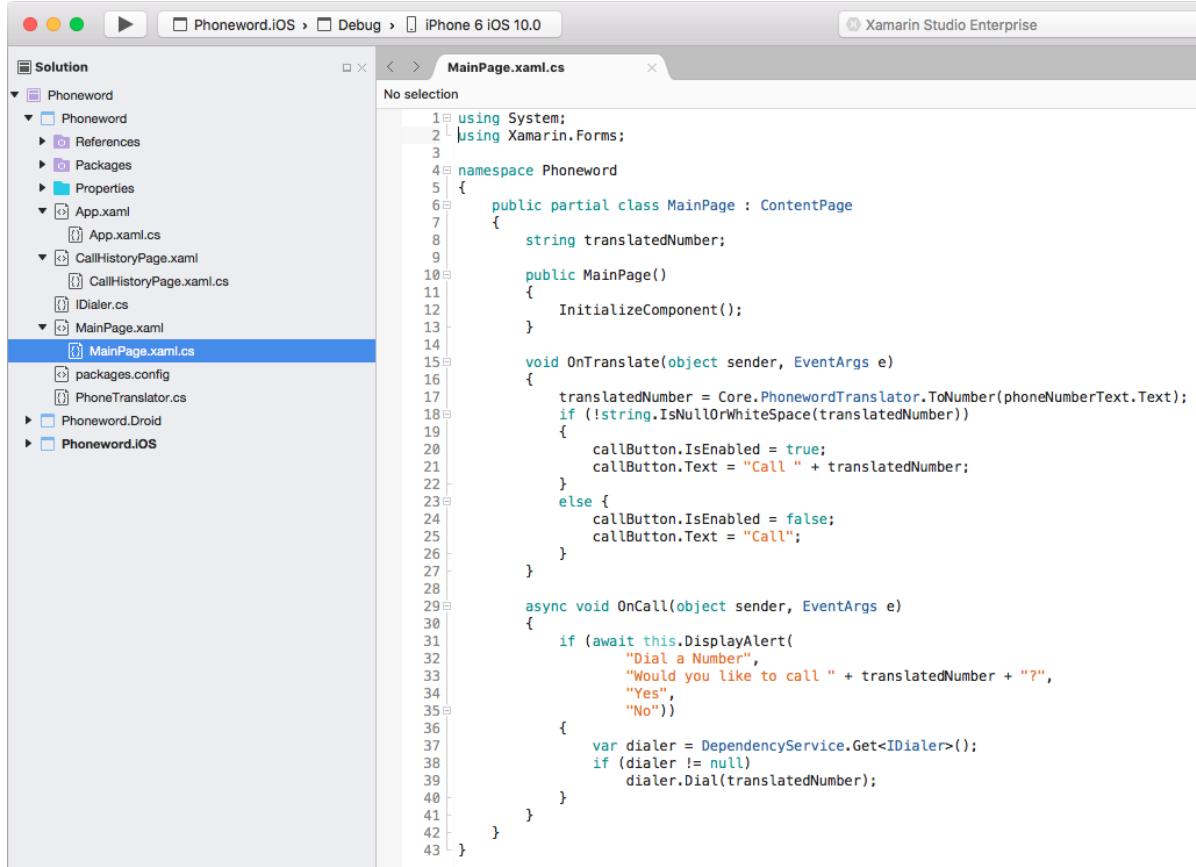
```

<StackLayout VerticalOptions="FillAndExpand"
            HorizontalOptions="FillAndExpand"
            Orientation="Vertical"
            Spacing="15">
  ...
  <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
  <Button x:Name="callHistoryButton" Text="Call History" IsEnabled="false"
          Clicked="OnCallHistory" />
</StackLayout>

```

Save the changes to **MainPage.xaml** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

10. In the **Solution Pad**, double-click **MainPage.xaml.cs** to open it:



11. In **MainPage.xaml.cs**, add the `OnCallHistory` event handler method, and modify the `OnCall` event handler method to add the translated phone number to the `App.PhoneNumbers` collection and enable the

`callHistoryButton`, provided that the `dialer` variable is not `null`:

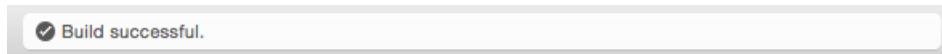
```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        ...
        ...
        ...
        if (dialer != null) {
            App.PhoneNumbers.Add (translatedNumber);
            callHistoryButton.IsEnabled = true;
            dialer.Dial (translatedNumber);
        }
        ...
    }

    async void OnCallHistory(object sender, EventArgs e)
    {
        await Navigation.PushAsync (new CallHistoryPage ());
    }
}
}
```

Save the changes to **MainPage.xaml.cs** by choosing **File > Save** (or by pressing **⌘ + S**), and close the file.

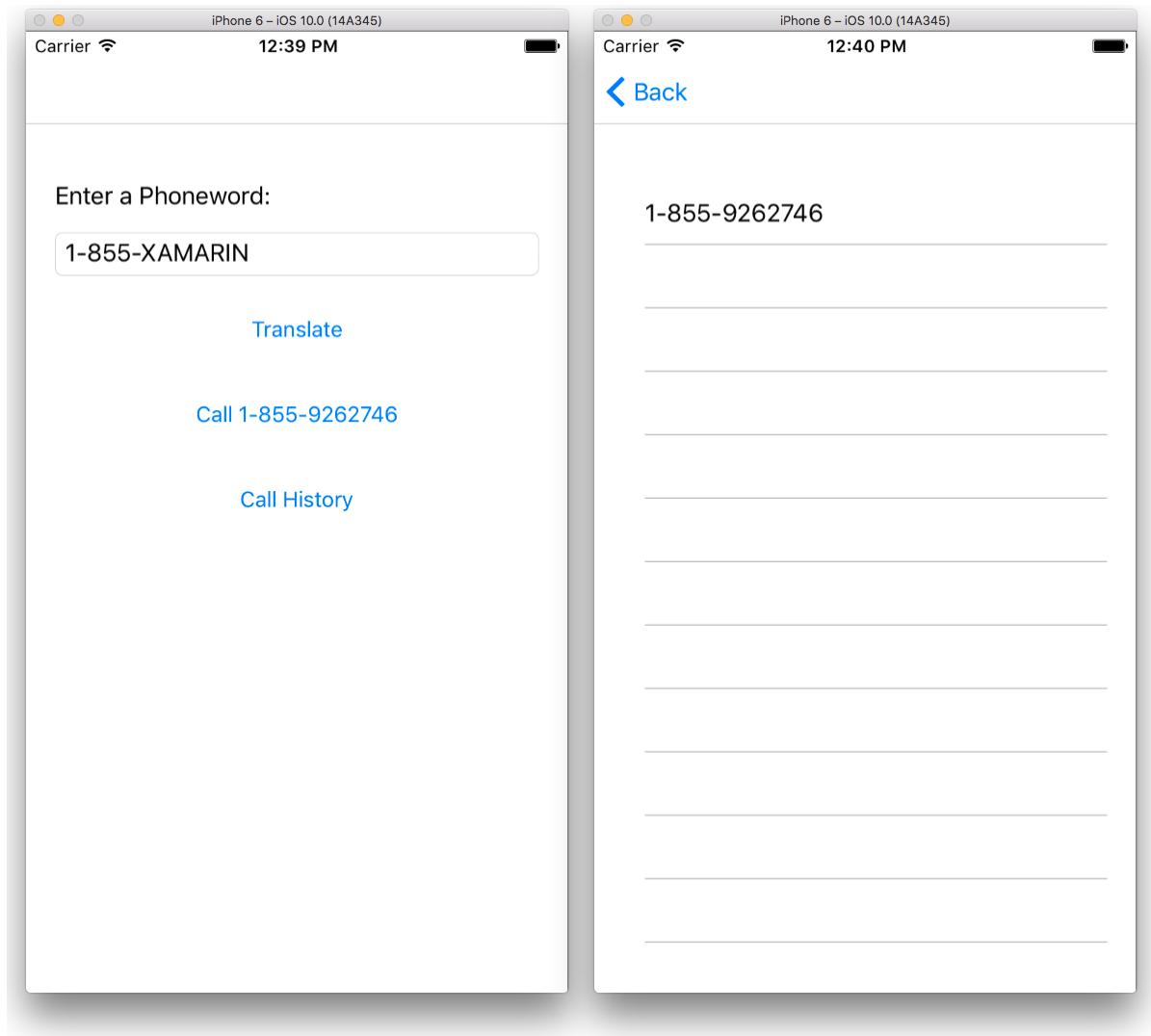
12. In Visual Studio for Mac, select the **Build > Build All** menu item (or press **⌘ + B**). The application will build and a success message will appear in the Visual Studio for Mac toolbar:



If there are errors, repeat the previous steps and correct any mistakes until the application builds successfully.

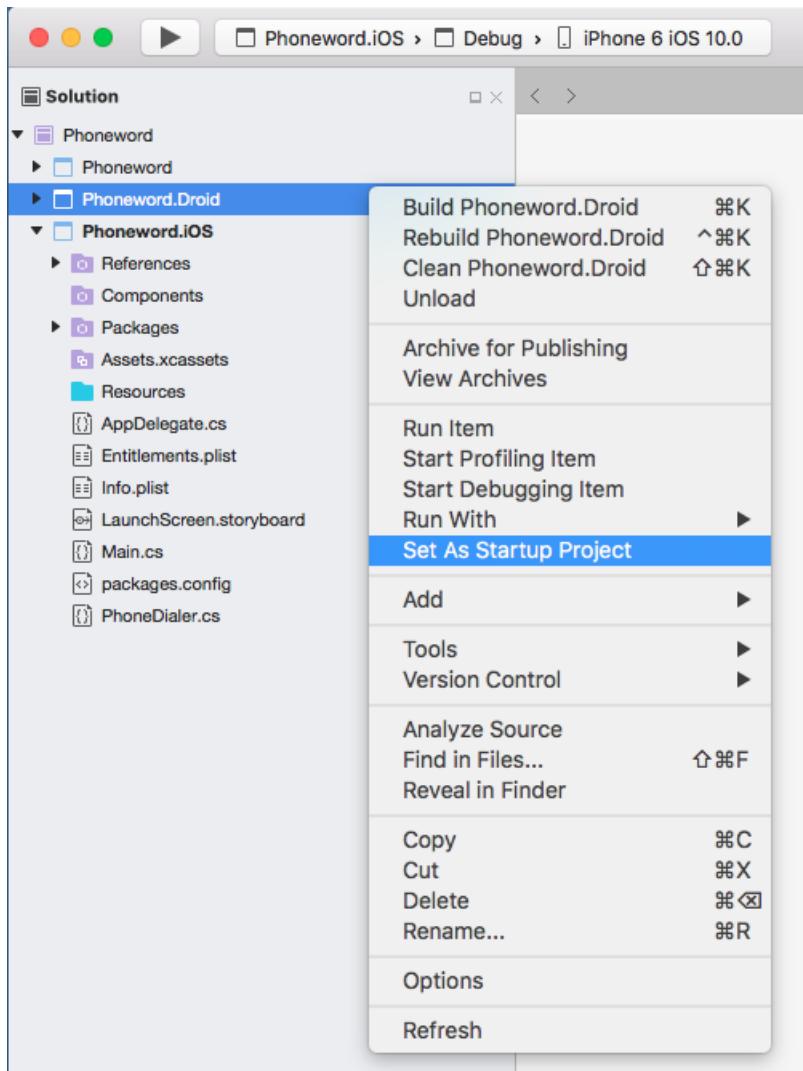
13. In the Visual Studio for Mac toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application inside the iOS Simulator:



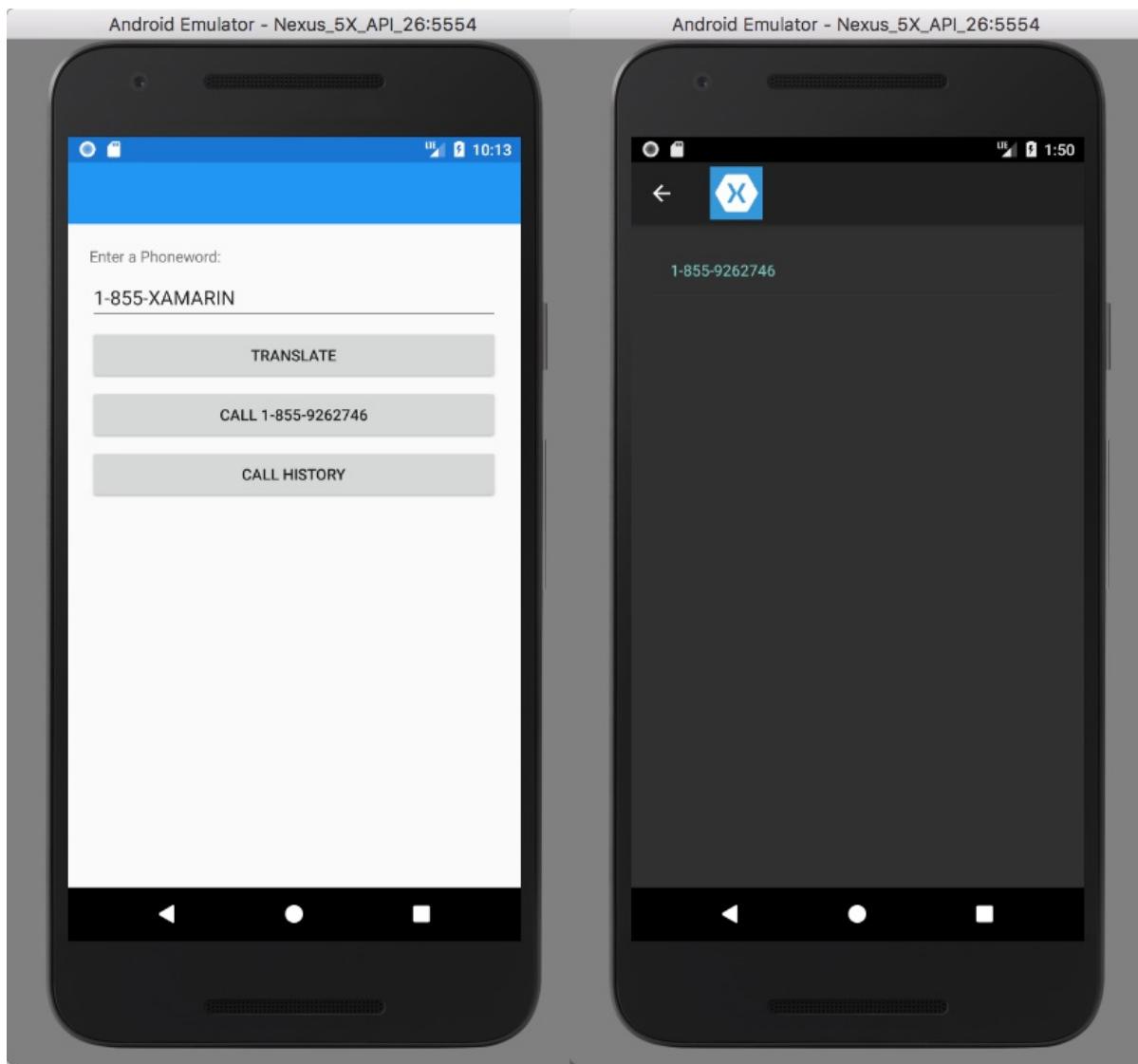


Note: phone calls are not supported in the iOS Simulator.

14. In the **Solution Pad**, select the **Phoneword.Droid** project, right-click and select **Set As Startup Project**:



15. In the Visual Studio for Mac toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application inside an Android emulator:



#### NOTE

Phone calls are not supported in device emulators.

Congratulations on completing a multiscreen Xamarin.Forms application. The [next topic](#) in this guide reviews the steps that were taken in this walkthrough to develop an understanding of page navigation and data binding using Xamarin.Forms.

## Related Links

- [Phoneword \(sample\)](#)
- [PhonewordMultiscreen \(sample\)](#)

# Xamarin.Forms Multiscreen Deep Dive

7/12/2018 • 3 minutes to read • [Edit Online](#)

In the [Xamarin.Forms Multiscreen Quickstart](#), the Phoneword application was extended to include a second screen that keeps track of the call history for the application. This article reviews what has been built, to develop an understanding of page navigation and data binding in a Xamarin.Forms application.

## Navigation

Xamarin.Forms provides a built-in navigation model that manages the navigation and user-experience of a stack of pages. This model implements a last-in, first-out (LIFO) stack of `Page` objects. To move from one page to another an application will push a new page onto this stack. To return back to the previous page the application will pop the current page from the stack.

Xamarin.Forms has a `NavigationPage` class that manages the stack of `Page` objects. The `NavigationPage` class will also add a navigation bar to the top of the page that displays a title and a platform-appropriate Back button that will return to the previous page. The following code example shows how to wrap a `NavigationPage` around the first page in an application:

```
public App ()  
{  
    ...  
    MainPage = new NavigationPage (new MainPage ());  
}
```

All `ContentPage` instances have a `Navigation` property that exposes methods to modify the page stack. These methods should only be invoked if the application includes a `NavigationPage`. To navigate to the `CallHistoryPage`, it is necessary to invoke the `PushAsync` method as demonstrated in the code example below:

```
async void OnCallHistory(object sender, EventArgs e)  
{  
    await Navigation.PushAsync (new CallHistoryPage ());  
}
```

This causes the new `CallHistoryPage` object to be pushed onto the Navigation stack. To programmatically return back to the original page, the `CallHistoryPage` object must invoke the `PopAsync` method, as demonstrated in the code example below:

```
await Navigation.PopAsync();
```

However, in the Phoneword application this code isn't required as the `NavigationPage` class adds a navigation bar to the top of the page that includes a platform appropriate Back button that will return to the previous page.

## Data Binding

Data binding is used to simplify how a Xamarin.Forms application displays and interacts with its data. It establishes a connection between the user interface and the underlying application. The `BindableObject` class contains much of the infrastructure to support data binding.

Data binding defines the relationship between two objects. The *source* object will provide data. The *target* object

will consume (and often display) the data from the source object. In the Phoneword application, the binding target is the `ListView` control that displays phone numbers, while the `PhoneNumbers` collection is the binding source.

The `PhoneNumbers` collection is declared and initialized in the `App` class, as shown in the following code example:

```
public partial class App : Application
{
    public static List<string> PhoneNumbers { get; set; }

    public App ()
    {
        PhoneNumbers = new List<string>();
        ...
    }
    ...
}
```

The `ListView` instance is declared and initialized in the `callHistoryPage` class, as shown in the following code example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage ...>
    xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    ...
    <ContentPage.Content>
        ...
        <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
        ...
    </ContentPage.Content>
</ContentPage>
```

In this example, the `ListView` control will display the `IEnumerable` collection of data that the `ItemsSource` property binds to. The collection of data can be objects of any type, but by default, `ListView` will use the `ToString` method of each item to display that item. The `x:Static` markup extension is used to indicate that the `ItemsSource` property will be bound to the static `PhoneNumbers` property of the `App` class, which can be located in the `local` namespace.

For more information about data binding, see [Data Binding Basics](#). For more information about XAML markup extensions, see [XAML Markup Extensions](#).

## Additional Concepts Introduced in Phoneword

The `ListView` is responsible for displaying a collection of items on the screen. Each item in the `ListView` is contained in a single cell. For more information about using the `ListView` control, see [ListView](#).

## Summary

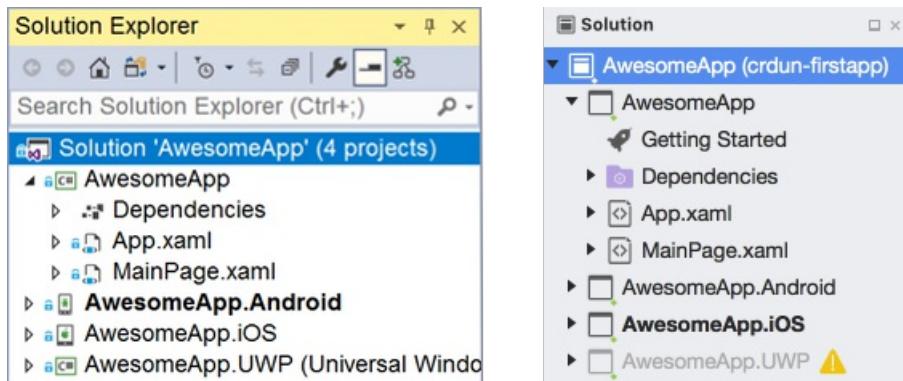
This article has introduced page navigation and data binding in a Xamarin.Forms application, and demonstrated their use in a multi-screen cross-platform application.

# An Introduction to Xamarin.Forms

10/9/2018 • 20 minutes to read • [Edit Online](#)

Xamarin.Forms is a framework that allows developers to build cross-platform applications for Android, iOS, and Windows. Code and user interface definitions are shared across platforms, but rendered with native controls. This article provides an introduction to Xamarin.Forms and how to get started writing applications with C# and XAML in Visual Studio.

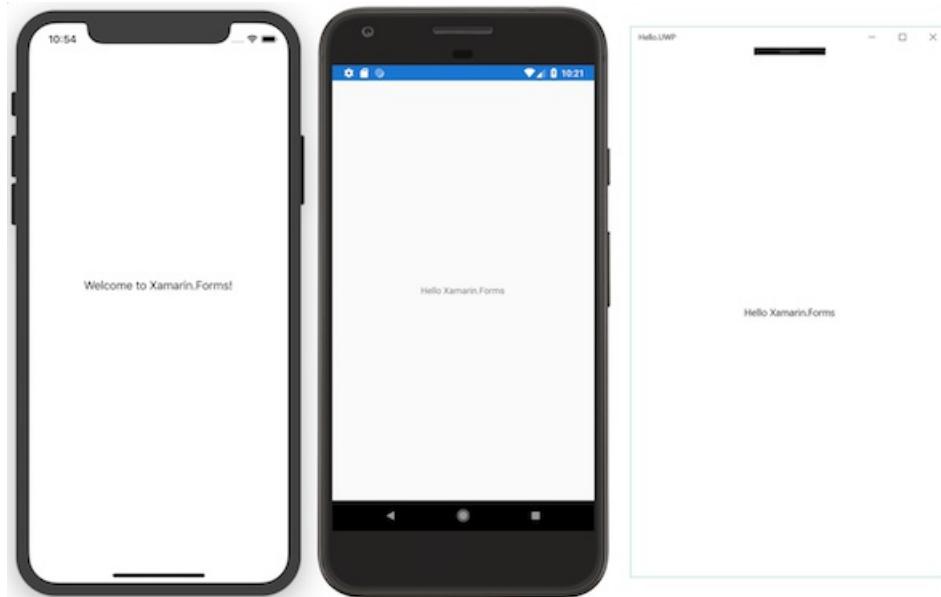
Xamarin.Forms applications use .NET Standard projects to contain the shared code, and separate application projects to consume the shared code and build the output required for each platform. When you create new Xamarin.Forms app, the solution will contain the shared code project (containing C# and XAML files) plus the platform-specific projects as shown in this screenshot:



When writing Xamarin.Forms apps, your code and user-interface will be added to the top, .NET Standard project which is referenced by the Android, iOS, and UWP projects. You will build and run the Android, iOS, and UWP projects to test and deploy your app.

## Examining a Xamarin.Forms application

The default Xamarin.Forms app template in Visual Studio displays a single text label. If you run the application, it should appear similar to the following screenshots:



Each screen in the screenshots corresponds to a *Page* in Xamarin.Forms. A [Page](#) represents an *Activity* in Android, a *View Controller* in iOS, or a *Page* in the Windows Universal Platform (UWP). The sample in the

screenshots above instantiates a `ContentPage` object and uses that to display a `Label`.

To maximize the reuse of the startup code, Xamarin.Forms applications have a single class named `App` that is responsible for instantiating the first `Page` that will be displayed. An example of the `App` class can be seen in the following code (in **App.xaml.cs**):

```
public partial class App : Application
{
    public App ()
    {
        InitializeComponent();
        MainPage = new MainPage(); // sets the App.MainPage property to an instance of the MainPage class
    }
}
```

This code instantiates a new `ContentPage` object called `MainPage` that will display a single `Label` centered both vertically and horizontally on the page. The XAML in the **MainPage.xaml** file looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:AwesomeApp"
    x:Class="AwesomeApp.MainPage">
    <StackLayout>
        <Label Text="Hello Xamarin.Forms"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

## Launching the initial Xamarin.Forms page on each platform

### TIP

The platform-specific information in this section is provided to help understand how Xamarin.Forms works. The project templates already include these classes; you are not required to code them yourself.

You can skip to the [user interface](#) section and read this part later.

To use a page (such as **MainPage** in the example above) inside an application, each platform application must initialize the Xamarin.Forms framework and provide an instance of the page as it is starting up. This initialization step varies from platform to platform and is discussed in the following sections.

### iOS

To launch the initial Xamarin.Forms page in iOS, the platform project includes the `AppDelegate` class that inherits from the `Xamarin.Forms.Platform.iOS.FormsApplicationDelegate` class, as demonstrated in the following code example:

```
[Register("AppDelegate")]
public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
{
    public override bool FinishedLaunching(UIApplication app, NSDictionary options)
    {
        global::Xamarin.Forms.Forms.Init ();
        LoadApplication (new App ());
        return base.FinishedLaunching (app, options);
    }
}
```

The `FinishedLaunching` override initializes the Xamarin.Forms framework by calling the `Init` method. This causes the iOS-specific implementation of Xamarin.Forms to be loaded in the application before the root view controller is set by the call to the `LoadApplication` method.

#### Android

To launch the initial Xamarin.Forms page in Android, the platform project includes code that creates an `Activity` with the `MainLauncher` attribute, with the activity inheriting from the `FormsAppCompatActivity` class, as demonstrated in the following code example:

```
namespace HelloXamarinFormsWorld.Android
{
    [Activity(Label = "HelloXamarinFormsWorld", Theme = "@style/MainTheme", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : AppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication (new App ());
        }
    }
}
```

The `OnCreate` override initializes the Xamarin.Forms framework by calling the `Init` method. This causes the Android-specific implementation of Xamarin.Forms to be loaded in the application before the Xamarin.Forms application is loaded.

#### Universal Windows Platform (UWP)

In Universal Windows Platform (UWP) applications, the `Init` method that initializes the Xamarin.Forms framework is invoked from the `App` class:

```
Xamarin.Forms.Forms.Init (e);

if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    ...
}
```

This causes the UWP-specific implementation of Xamarin.Forms to be loaded in the application. The initial Xamarin.Forms page is launched by the `MainPage` class, as demonstrated in the following code example:

```
public partial class MainPage
{
    public MainPage()
    {
        this.InitializeComponent();
        this.LoadApplication(new HelloXamarinFormsWorld.App());
    }
}
```

The Xamarin.Forms application is loaded with the `LoadApplication` method. Visual Studio adds all the above code when you create a new Xamarin.Forms project.

## User interface

There are two techniques to create user interfaces in Xamarin.Forms:

- Create the user-interface entirely with C# source code.
- *Extensible Application Markup Language* (XAML), a declarative markup language that is used to describe user interfaces.

The same results can be achieved regardless of which method you use (and both are explained below). For more information about Xamarin.Forms XAML, see [XAML Basics](#).

## Views and layouts

There are four main control groups used to create the user interface of a Xamarin.Forms application.

- **Pages** – Xamarin.Forms pages represent cross-platform mobile application screens. For more information about pages, see [Xamarin.Forms Pages](#).
- **Layouts** – Xamarin.Forms layouts are containers used to compose views into logical structures. For more information about layouts, see [Xamarin.Forms Layouts](#).
- **Views** – Xamarin.Forms views are the controls displayed on the user interface, such as labels, buttons, and text entry boxes. For more information about views, see [Xamarin.Forms Views](#).
- **Cells** – Xamarin.Forms cells are specialized elements used for items in a list, and describe how each item in a list should be drawn. For more information about cells, see [Xamarin.Forms Cells](#).

At runtime, each control will be mapped to its native equivalent, which is rendered on the screen.

Controls are hosted inside of a layout. The `StackLayout` class – a commonly used layout – is discussed below.

## StackLayout

The `StackLayout` simplifies cross-platform application development by automatically arranging controls on the screen regardless of the screen size. Each child element is positioned one after the other, either horizontally or vertically in the order they were added. How much space the `StackLayout` will use depends on how the `HorizontalOptions` and `VerticalOptions` properties are set, but by default the `StackLayout` will try to use the entire screen.

The following XAML code shows an example of using a `StackLayout` to arrange three `Label` controls:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample1" Padding="20">
  <StackLayout Spacing="10">
    <Label Text="Stop" BackgroundColor="Red" Font="20" />
    <Label Text="Slow down" BackgroundColor="Yellow" Font="20" />
    <Label Text="Go" BackgroundColor="Green" Font="20" />
  </StackLayout>
</ContentPage>
```

The equivalent C# code is shown in the following code example:

```

public class StackLayoutExample : ContentPage
{
    public StackLayoutExample()
    {
        Padding = new Thickness(20);
        var red = new Label
        {
            Text = "Stop", BackgroundColor = Color.Red, FontSize = 20
        };
        var yellow = new Label
        {
            Text = "Slow down", BackgroundColor = Color.Yellow, FontSize = 20
        };
        var green = new Label
        {
            Text = "Go", BackgroundColor = Color.Green, FontSize = 20
        };

        Content = new StackLayout
        {
            Spacing = 10,
            Children = { red, yellow, green }
        };
    }
}

```

By default the `StackLayout` assumes a vertical orientation as shown in the following screenshots:



The orientation of the `StackLayout` can be changed to a horizontal orientation, as demonstrated in the following XAML code example:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample2" Padding="20">
    <StackLayout Spacing="10" VerticalOptions="End" Orientation="Horizontal" HorizontalOptions="Start">
        <Label Text="Stop" BackgroundColor="Red" Font="20" />
        <Label Text="Slow down" BackgroundColor="Yellow" Font="20" />
        <Label Text="Go" BackgroundColor="Green" Font="20" />
    </StackLayout>
</ContentPage>

```

The equivalent C# code is shown in the following code example:

```

public class StackLayoutExample: ContentPage
{
    public StackLayoutExample()
    {
        // Code that creates red, yellow, green labels removed for clarity (see above)
        Content = new StackLayout
        {
            Spacing = 10,
            VerticalOptions = LayoutOptions.End,
            Orientation = StackOrientation.Horizontal,
            HorizontalOptions = LayoutOptions.Start,
            Children = { red, yellow, green }
        };
    }
}

```

The following screenshots show the resulting layout:



The size of controls can be set through the `HeightRequest` and `WidthRequest` properties, as demonstrated in the following XAML code example:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample3" Padding="20">
    <StackLayout Spacing="10" VerticalOptions="End" Orientation="Horizontal" HorizontalOptions="Start">
        <Label Text="Stop" BackgroundColor="Red" Font="20" WidthRequest="100" />
        <Label Text="Slow down" BackgroundColor="Yellow" Font="20" WidthRequest="100" />
        <Label Text="Go" BackgroundColor="Green" Font="20" WidthRequest="200" />
    </StackLayout>
</ContentPage>

```

The equivalent C# code is shown in the following code example:

```

var red = new Label
{
    Text = "Stop", BackgroundColor = Color.Red, FontSize = 20, WidthRequest = 100
};
var yellow = new Label
{
    Text = "Slow down", BackgroundColor = Color.Yellow, FontSize = 20, WidthRequest = 100
};
var green = new Label
{
    Text = "Go", BackgroundColor = Color.Green, FontSize = 20, WidthRequest = 200
};

Content = new StackLayout
{
    Spacing = 10,
    VerticalOptions = LayoutOptions.End,
    Orientation = StackOrientation.Horizontal,
    HorizontalOptions = LayoutOptions.Start,
    Children = { red, yellow, green }
};

```

The following screenshots show the resulting layout:



For more information about the [StackLayout](#) class, see [StackLayout](#).

## Lists in Xamarin.Forms

The [ListView](#) control is responsible for displaying a collection of items on the screen – each item in the [ListView](#) will be contained in a single cell. By default, a [ListView](#) will use the built-in [TextCell](#) template and render a single line of text.

The following code example shows a simple [ListView](#) example:

```

var listView = new ListView
{
    RowHeight = 40
};
listView.ItemsSource = new string []
{
    "Buy pears", "Buy oranges", "Buy mangos", "Buy apples", "Buy bananas"
};
Content = new StackLayout
{
    VerticalOptions = LayoutOptions.FillAndExpand,
    Children = { listView }
};

```

The following screenshot shows the resulting [ListView](#):



For more information about the [ListView](#) control, see [ListView](#).

### Binding to a custom class

The [ListView](#) control can also display custom objects using the default [TextCell](#) template.

The following code example shows the [TodoItem](#) class:

```

public class TodoItem
{
    public string Name { get; set; }
    public bool Done { get; set; }
}

```

The [ListView](#) control can be populated as demonstrated in the following code example:

```

listView.ItemsSource = new TodoItem [] {
    new TodoItem { Name = "Buy pears" },
    new TodoItem { Name = "Buy oranges", Done=true},
    new TodoItem { Name = "Buy mangos" },
    new TodoItem { Name = "Buy apples", Done=true },
    new TodoItem { Name = "Buy bananas", Done=true }
};

```

A binding can be created to set which [TodoItem](#) property is displayed by the [ListView](#), as demonstrated in the following code example:

```

listView.ItemTemplate = new DataTemplate(typeof(TextCell));
listView.ItemTemplate.SetBinding(TextCell.TextProperty, "Name");

```

This creates a binding that specifies the path to the [TodoItem.Name](#) property, and will result in the previously displayed screenshot.

For more information about binding to a custom class, see [ListView Data Sources](#).

### Selecting an item in a ListView

To respond to a user touching a cell in a `ListView`, the `ItemSelected` event should be handled, as demonstrated in the following code example:

```
listView.ItemSelected += async (sender, e) => {
    await DisplayAlert("Tapped!", e.SelectedItem + " was tapped.", "OK");
};
```

When contained within a `NavigationPage`, the `PushAsync` method can be used to open a new page with built-in back-navigation. The `ItemSelected` event can access the object that was associated with the cell through the `e.SelectedItem` property, bind it to a new page and display the new page using `PushAsync`, as demonstrated in the following code example:

```
listView.ItemSelected += async (sender, e) => {
    var todoItem = (TodoItem)e.SelectedItem;
    var todoPage = new TodoItemPage(todoItem); // so the new page shows correct data
    await Navigation.PushAsync(todoPage);
};
```

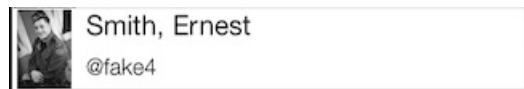
Each platform implements built-in back-navigation in its own way. For more information, see [Navigation](#).

For more information about `ListView` selection, see [ListView Interactivity](#).

### Customizing the appearance of a cell

Cell appearance can be customized by subclassing the `ViewCell` class, and setting the type of this class to the `ItemTemplate` property of the `ListView`.

The cell shown in the following screenshot is composed of one `Image` and two `Label` controls:



To create this custom layout, the `ViewCell` class should be subclassed, as demonstrated in the following code example:

```

class EmployeeCell : ViewCell
{
    public EmployeeCell()
    {
        var image = new Image
        {
            HorizontalOptions = LayoutOptions.Start
        };
        image.SetBinding(Image.SourceProperty, new Binding("ImageUri"));
        image.WidthRequest = image.HeightRequest = 40;

        var nameLayout = CreateNameLayout();
        var viewLayout = new StackLayout()
        {
            Orientation = StackOrientation.Horizontal,
            Children = { image, nameLayout }
        };
        View = viewLayout;
    }

    static StackLayout CreateNameLayout()
    {
        var nameLabel = new Label
        {
            HorizontalOptions= LayoutOptions.FillAndExpand
        };
        nameLabel.SetBinding(Label.TextProperty, "DisplayName");

        var twitterLabel = new Label
        {
            HorizontalOptions = LayoutOptions.FillAndExpand,
            Font = Fonts.Twitter
        };
        twitterLabel.SetBinding(Label.TextProperty, "Twitter");

        var nameLayout = new StackLayout()
        {
            HorizontalOptions = LayoutOptions.StartAndExpand,
            Orientation = StackOrientation.Vertical,
            Children = { nameLabel, twitterLabel }
        };
        return nameLayout;
    }
}

```

The code performs the following tasks:

- It adds an `Image` control and binds it to the `ImageUri` property of the `Employee` object. For more information about data binding, see [Data Binding](#).
- It creates a `StackLayout` with a vertical orientation to hold the two `Label` controls. The `Label` controls are bound to the `DisplayName` and the `Twitter` properties of the `Employee` object.
- It creates a `StackLayout` that will host the existing `Image` and `StackLayout`. It will arrange its children using a horizontal orientation.

Once the custom cell has been created it can be consumed by a `ListView` control by wrapping it in a `DataTemplate`, as demonstrated in the following code example:

```

List<Employee> myListOfEmployeeObjects = GetAListOfAllEmployees();
var listView = new ListView
{
    RowHeight = 40
};
listView.ItemsSource = myListOfEmployeeObjects;
listView.ItemTemplate = new DataTemplate(typeof(EmployeeCell));

```

This code will provide a `List` of `Employee` to the `ListView`. Each cell will be rendered using the `EmployeeCell` class. The `ListView` will pass the `Employee` object to the `EmployeeCell` as its `BindingContext`.

For more information about customizing cell appearance, see [Cell Appearance](#).

## Using XAML to create and customize a list

The XAML equivalent of the `ListView` in the previous section is demonstrated in the following code example:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:XamarinFormsXamlSample;assembly=XamarinFormsXamlSample"
    xmlns:constants="clr-namespace:XamarinFormsSample;assembly=XamarinFormsXamlSample"
    x:Class="XamarinFormsXamlSample.Views.EmployeeListPage"
    Title="Employee List">
    <ListView x:Name="listView" IsVisible="false" ItemsSource="{x:Static local:App.Employees}"
    ItemSelected="EmployeeListOnItemSelected">
        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <ViewCell.View>
                        <StackLayout Orientation="Horizontal">
                            <Image Source="{Binding ImageUri}" WidthRequest="40" HeightRequest="40" />
                            <StackLayout Orientation="Vertical" HorizontalOptions="StartAndExpand">
                                <Label Text="{Binding DisplayName}" HorizontalOptions="FillAndExpand" />
                                <Label Text="{Binding Twitter}" Font="{x:Static constants:Fonts.Twitter}"/>
                            </StackLayout>
                        </StackLayout>
                    </ViewCell.View>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>

```

This XAML defines a `ContentPage` that contains a `ListView`. The data source of the `ListView` is set via the `ItemsSource` attribute. The layout of each row in the `ItemsSource` is defined within the `ListView.ItemTemplate` element.

## Data binding

Data binding connects two objects, called the *source* and the *target*. The *source* object provides the data. The *target* object will consume (and often display) data from the source object. For example, a `Label` (*target* object) will commonly bind its `Text` property to a public `string` property in a *source* object. The following diagram illustrates the binding relationship:



The main benefit of data binding is that you no longer have to worry about synchronizing data between your views and data source. Changes in the *source* object are automatically pushed to the *target* object behind-the-scenes by the binding framework, and changes in the target object can be optionally pushed back to the *source* object.

Establishing data binding is a two step process:

- The `BindingContext` property of the *target* object must be set to the *source*.
- A binding must be established between the *target* and the *source*. In XAML, this is achieved by using the `Binding` markup extension. In C#, this is achieved by the `SetBinding` method.

For more information about data binding, see [Data Binding Basics](#).

## XAML

The following code shows an example of performing data binding in XAML:

```
<Entry Text="{Binding FirstName}" ... />
```

A binding between the `Entry.Text` property and the `FirstName` property of the *source* object is established. Changes made in the `Entry` control will automatically be propagated to the `employeeToDisplay` object. Similarly, if changes are made to the `employeeToDisplay.FirstName` property, the Xamarin.Forms binding engine will also update the contents of the `Entry` control. This is known as a *two-way binding*. In order for two-way binding to work, the model class must implement the `INotifyPropertyChanged` interface.

Although the `BindingContext` property of the `EmployeeDetailPage` class can be set in XAML, here it's set in code-behind to an instance of an `Employee` object:

```
public EmployeeDetailPage(Employee employee)
{
    InitializeComponent();
    this.BindingContext = employee;
}
```

While the `BindingContext` property of each *target* object can be individually set, this isn't necessary. `BindingContext` is a special property that's inherited by all its children. Therefore, when the `BindingContext` on the `ContentPage` is set to an `Employee` instance, all of the children of the `ContentPage` have the same `BindingContext`, and can bind to public properties of the `Employee` object.

## C#

The following code shows an example of performing data binding in C#:

```
public EmployeeDetailPage(Employee employeeToDisplay)
{
    this.BindingContext = employeeToDisplay;
    var firstName = new Entry()
    {
        HorizontalOptions = LayoutOptions.FillAndExpand
    };
    firstName.SetBinding(Entry.TextProperty, "FirstName");
    ...
}
```

The `ContentPage` constructor is passed an instance of an `Employee` object, and sets the `BindingContext` to the object to bind to. An `Entry` control is instantiated, and the binding between the `Entry.Text` property and the `FirstName` property of the *source* object is set. Changes made in the `Entry` control will automatically be

propagated to the `employeeToDisplay` object. Similarly, if changes are made to the `employeeToDisplay.FirstName` property, the Xamarin.Forms binding engine will also update the contents of the `Entry` control. This is known as a *two-way binding*. In order for two-way binding to work, the model class must implement the `INotifyPropertyChanged` interface.

The `SetBinding` method takes two parameters. The first parameter specifies information about the type of binding. The second parameter is used to provide information about what to bind to or how to bind. The second parameter is, in most cases, just a string holding the name of property on the `BindingContext`. The following syntax is used to bind to the `BindingContext` directly:

```
someLabel.SetBinding(Label.TextProperty, new Binding("."));
```

The dot syntax tells Xamarin.Forms to use the `BindingContext` as the data source instead of a property on the `BindingContext`. This is useful when the `BindingContext` is a simple type, such as a `string` or an `int`.

## Property change notification

By default, the *target* object only receives the value of the *source* object when the binding is created. To keep the UI synchronized with the data source, there must be a way to notify the *target* object when the *source* object has changed. This mechanism is provided by the `INotifyPropertyChanged` interface. Implementing this interface will provide notifications to any data-bound controls when the underlying property value changes.

Objects that implement `INotifyPropertyChanged` must raise the `PropertyChanged` event when one of their properties is updated with a new value, as demonstrated in the following code example:

```
public class MyObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set
        {
            if (value.Equals(_firstName, StringComparison.Ordinal))
            {
                // Nothing to do - the value hasn't changed;
                return;
            }
            _firstName = value;
            OnPropertyChanged();
        }
    }

    void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

When the `Myobject.FirstName` property changes, the `OnPropertyChanged` method is invoked, which will raise the `PropertyChanged` event. To avoid unnecessary events firing, the `PropertyChanged` event isn't raised if the property value doesn't change.

Note that in the `OnPropertyChanged` method the `propertyName` parameter is adorned with the `CallerMemberName` attribute. This ensures that if the `OnPropertyChanged` method is invoked with a `null` value, the `CallerMemberName` attribute will provide the name of the method that invoked `OnPropertyChanged`.

## Navigation

Xamarin.Forms provides a number of different page navigation experiences, depending upon the `Page` type being used. For `ContentPage` instances there are two navigation experiences:

- [Hierarchical Navigation](#)
- [Modal Navigation](#)

The `CarouselPage`, `MasterDetailPage` and `TabbedPage` classes provide alternative navigation experiences. For more information, see [Navigation](#).

### Hierarchical navigation

The `NavigationPage` class provides a hierarchical navigation experience where the user is able to navigate through pages, forwards and backwards, as desired. The class implements navigation as a last-in, first-out (LIFO) stack of `Page` objects.

In hierarchical navigation, the `NavigationPage` class is used to navigate through a stack of `ContentPage` objects. To move from one page to another, an application will push a new page onto the navigation stack, where it will become the active page. To return back to the previous page, the application will pop the current page from the navigation stack, and the new topmost page becomes the active page.

The first page added to a navigation stack is referred to as the *root* page of the application, and the following code example shows how this is accomplished:

```
public App ()  
{  
    MainPage = new NavigationPage(new EmployeeListPage());  
}
```

To navigate to the `LoginPage`, it is necessary to invoke the `PushAsync` method on the `Navigation` property of the current page, as demonstrated in the following code example:

```
await Navigation.PushAsync(new LoginPage());
```

This causes the new `LoginPage` object to be pushed on the navigation stack, where it becomes the active page.

The active page can be popped from the navigation stack by pressing the *Back* button on the device, regardless of whether this is a physical button on the device or an on-screen button. To programmatically return to the previous page, the `LoginPage` instance must invoke the `PopAsync` method, as demonstrated in the following code example:

```
await Navigation.PopAsync();
```

For more information about hierarchical navigation, see [Hierarchical Navigation](#).

### Modal navigation

Xamarin.Forms provides support for modal pages. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled.

A modal page can be any of the `Page` types supported by Xamarin.Forms. To display a modal page the

application will push it onto the navigation stack, where it will become the active page. To return to the previous page the application will pop the current page from the navigation stack, and the new topmost page becomes the active page.

Modal navigation methods are exposed by the `Navigation` property on any `Page` derived types. The `Navigation` property also exposes a `ModalStack` property from which the modal pages in the navigation stack can be obtained. However, there is no concept of performing modal stack manipulation, or popping to the root page in modal navigation. This is because these operations are not universally supported on the underlying platforms.

#### NOTE

A `NavigationPage` instance is not required for performing modal page navigation.

To modally navigate to the `LoginPage` it is necessary to invoke the `PushModalAsync` method on the `Navigation` property of the current page, as demonstrated in the following code example:

```
await Navigation.PushModalAsync(new LoginPage());
```

This causes the `LoginPage` instance to be pushed onto the navigation stack, where it becomes the active page.

The active page can be popped from the navigation stack by pressing the *Back* button on the device, regardless of whether this is a physical button on the device or an on-screen button. To programmatically return to the original page, the `LoginPage` instance must invoke the `PopModalAsync` method, as demonstrated in the following code example:

```
await Navigation.PopModalAsync();
```

This causes the `LoginPage` instance to be removed from the navigation stack, with the new topmost page becoming the active page.

For more information about modal navigation, see [Modal Pages](#).

## Next steps

This introductory article should enable you to start writing Xamarin.Forms applications. Suggested next steps include reading about the following functionality:

- Control templates provide the ability to easily theme and re-theme application pages at runtime. For more information, see [Control Templates](#).
- Data templates provide the ability to define the presentation of data on supported controls. For more information, see [Data Templates](#).
- Shared code can access native functionality through the `DependencyService` class. For more information, see [Accessing Native Features with DependencyService](#).
- Xamarin.Forms includes a simple messaging service to send and receive messages, therefore reducing coupling between classes. For more information, see [Publish and Subscribe with MessagingCenter](#).
- Each page, layout, and control is rendered differently on each platform using a `Renderer` class that in turn creates a native control, arranges it on the screen, and adds the behavior specified in the shared code. Developers can implement their own custom `Renderer` classes to customize the appearance and/or behavior of a control. For more information, see [Custom Renderers](#).
- Effects also allow the native controls on each platform to be customized. Effects are created in platform-specific projects by subclassing the `PlatformEffect` control, and are consumed by attaching them to an

appropriate Xamarin.Forms control. For more information, see [Effects](#).

Alternatively, [\*Creating Mobile Apps with Xamarin.Forms\*](#), a book by Charles Petzold, is a good place to learn more about Xamarin.Forms. The book is available as a PDF or in a variety of ebook formats.

## Related links

- [XAML Basics](#)
- [Controls Reference](#)
- [User Interface](#)
- [Xamarin.Forms Samples](#)
- [Getting Started Samples](#)
- [Xamarin.Forms API reference](#)
- [Free Self-Guided Learning \(video\)](#)

# eXtensible Application Markup Language (XAML)

11/12/2018 • 2 minutes to read • [Edit Online](#)

XAML is a declarative markup language that can be used to define user interfaces. The user interface is defined in an XML file using the XAML syntax, while runtime behavior is defined in a separate code-behind file.

## Evolve 2016: Becoming a XAML Master

### NOTE

Try out the [XAML Standard Preview](#)

## XAML Basics

XAML allows developers to define user interfaces in Xamarin.Forms applications using markup rather than code. XAML is never required in a Xamarin.Forms program but it is toolable, and is often more visually coherent and more succinct than equivalent code. XAML is particularly well suited for use with the popular Model-View-ViewModel (MVVM) application architecture: XAML defines the View that is linked to ViewModel code through XAML-based data bindings.

## XAML Compilation

XAML can be optionally compiled directly into intermediate language (IL) with the XAML compiler (XAMLC). This article describes how to use XAMLC, and its benefits.

## XAML Previewer

The [XAML Previewer](#) renders a live preview of a page side-by-side with the XAML markup, allowing you to see your user interface rendered as you type.

## XAML Namespaces

XAML uses the `xmlns` XML attribute for namespace declarations. This article introduces the XAML namespace syntax, and demonstrates how to declare a XAML namespace to access a type.

## XAML Markup Extensions

XAML includes markup extensions for setting attributes to values or objects beyond what can be expressed with simple strings. These include referencing constants, static properties and fields, resource dictionaries, and data bindings.

## Field Modifiers

The `x:FieldModifier` namespace attribute specifies the access level for generated fields for named XAML elements.

## Passing Arguments

XAML can be used to pass arguments to non-default constructors or to factory methods. This article demonstrates

using the XAML attributes that can be used to pass arguments to constructors, to call factory methods, and to specify the type of a generic argument.

## Bindable Properties

In Xamarin.Forms, the functionality of common language runtime (CLR) properties is extended by bindable properties. A bindable property is a special type of property, where the property's value is tracked by the Xamarin.Forms property system. This article provides an introduction to bindable properties, and demonstrates how to create and consume them.

## Attached Properties

An attached property is a special type of bindable property, defined in one class but attached to other objects, and recognizable in XAML as an attribute that contains a class and a property name separated by a period. This article provides an introduction to attached properties, and demonstrates how to create and consume them.

## Resource Dictionaries

XAML resources are definitions of objects that can be used more than once. A `ResourceDictionary` allows resources to be defined in a single location, and re-used throughout a Xamarin.Forms application. This article demonstrates how to create and consume a `ResourceDictionary`, and how to merge one `ResourceDictionary` into another.

# Xamarin.Forms XAML Basics

11/12/2018 • 3 minutes to read • [Edit Online](#)

XAML—the eXtensible Application Markup Language—allows developers to define user interfaces in Xamarin.Forms applications using markup rather than code. XAML is never required in a Xamarin.Forms program, but it is often more succinct and more visually coherent than equivalent code, and potentially toolable. XAML is particularly well suited for use with the popular MVVM (Model-View-ViewModel) application architecture: XAML defines the View that is linked to ViewModel code through XAML-based data bindings.

## XAML Basics Contents

- [Overview](#)
- [Part 1. Getting Started with XAML](#)
- [Part 2. Essential XAML Syntax](#)
- [Part 3. XAML Markup Extensions](#)
- [Part 4. Data Binding Basics](#)
- [Part 5. From Data Binding to MVVM](#)

In addition to these XAML Basics articles, you can download chapters of the book [Creating Mobile Apps with Xamarin.Forms](#):



XAML topics are covered in more depth in many chapters of the book, including:

<a href="#">Chapter 7. XAML vs. Code</a>	<a href="#">Download PDF</a>	<a href="#">Summary</a>
<a href="#">Chapter 8. Code and XAML in Harmony</a>	<a href="#">Download PDF</a>	<a href="#">Summary</a>
<a href="#">Chapter 10. XAML Markup Extensions</a>	<a href="#">Download PDF</a>	<a href="#">Summary</a>
<a href="#">Chapter 18. MVVM</a>	<a href="#">Download PDF</a>	<a href="#">Summary</a>

These chapters can be [downloaded for free](#).

## Overview

XAML is an XML-based language created by Microsoft as an alternative to programming code for instantiating and initializing objects, and organizing those objects in parent-child hierarchies. XAML has been adapted to several technologies within the .NET framework, but it has found its greatest utility in defining the layout of user interfaces within the Windows Presentation Foundation (WPF), Silverlight, the Windows Runtime, and the Universal Windows Platform (UWP).

XAML is also part of Xamarin.Forms, the cross-platform natively-based programming interface for iOS, Android,

and UWP mobile devices. Within the XAML file, the Xamarin.Forms developer can define user interfaces using all the Xamarin.Forms views, layouts, and pages, as well as custom classes. The XAML file can be either compiled or embedded in the executable. Either way, the XAML information is parsed at build time to locate named objects, and again at runtime to instantiate and initialize objects, and to establish links between these objects and programming code.

XAML has several advantages over equivalent code:

- XAML is often more succinct and readable than equivalent code.
- The parent-child hierarchy inherent in XML allows XAML to mimic with greater visual clarity the parent-child hierarchy of user-interface objects.
- XAML can be easily hand-written by programmers, but also lends itself to be toolable and generated by visual design tools.

Of course, there are also disadvantages, mostly related to limitations that are intrinsic to markup languages:

- XAML cannot contain code. All event handlers must be defined in a code file.
- XAML cannot contain loops for repetitive processing. (However, several Xamarin.Forms visual objects—most notably `ListView`—can generate multiple children based on the objects in its `ItemsSource` collection.)
- XAML cannot contain conditional processing (However, a data-binding can reference a code-based binding converter that effectively allows some conditional processing.)
- XAML generally cannot instantiate classes that do not define a parameterless constructor. (However, there is sometimes a way around this restriction.)
- XAML generally cannot call methods. (Again, this restriction can sometimes be overcome.)

There is not yet a visual designer for generating XAML in Xamarin.Forms applications. All XAML must be hand-written, but there is a [XAML Previewer](#). Programmers new to XAML might want to frequently build and run their applications, particularly after anything that might not be obviously correct. Even developers with lots of experience in XAML know that experimentation is rewarding.

XAML is basically XML, but XAML has some unique syntax features. The most important are:

- Property elements
- Attached properties
- Markup extensions

These features are *not* XML extensions. XAML is entirely legal XML. But these XAML syntax features use XML in unique ways. They are discussed in detail in the articles below, which conclude with an introduction to using XAML for implementing MVVM.

## Requirements

This article assumes a working familiarity with Xamarin.Forms. Reading [An Introduction to Xamarin.Forms](#) is highly recommended.

This article also assumes some familiarity with XML, including understanding the use of XML namespace declarations, and the terms *element*, *tag*, and *attribute*.

When you're familiar with Xamarin.Forms and XML, start reading [Part 1. Getting Started with XAML](#).

## Related Links

- [XamlSamples](#)
- [An Introduction to Xamarin.Forms](#)
- [Creating Mobile Apps book](#)

- [Xamarin.Forms Samples](#)

# Part 1. Getting Started with XAML

11/20/2018 • 15 minutes to read • [Edit Online](#)

In a Xamarin.Forms application, XAML is mostly used to define the visual contents of a page and works together with a C# code-behind file.

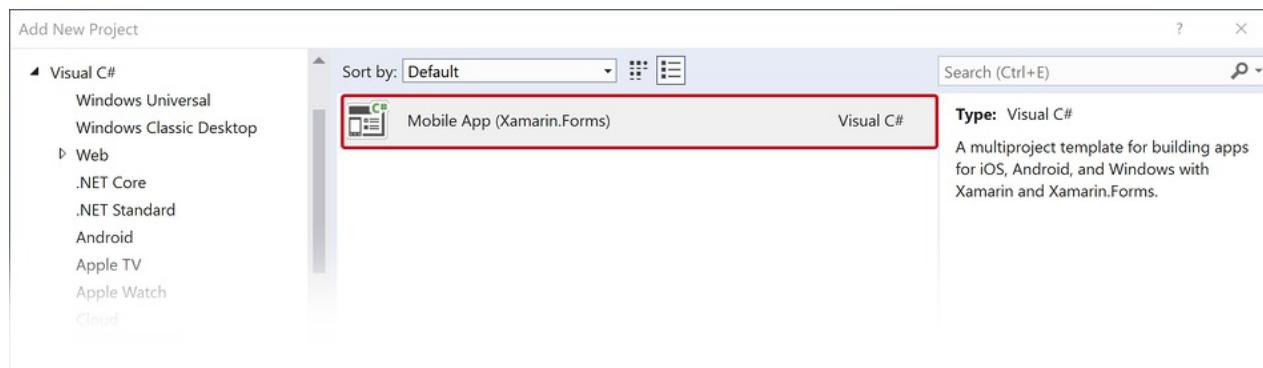
The code-behind file provides code support for the markup. Together, these two files contribute to a new class definition that includes child views and property initialization. Within the XAML file, classes and properties are referenced with XML elements and attributes, and links between the markup and code are established.

## Creating the Solution

To begin editing your first XAML file, use Visual Studio or Visual Studio for Mac to create a new Xamarin.Forms solution. (Select the tab below corresponding to your environment.)

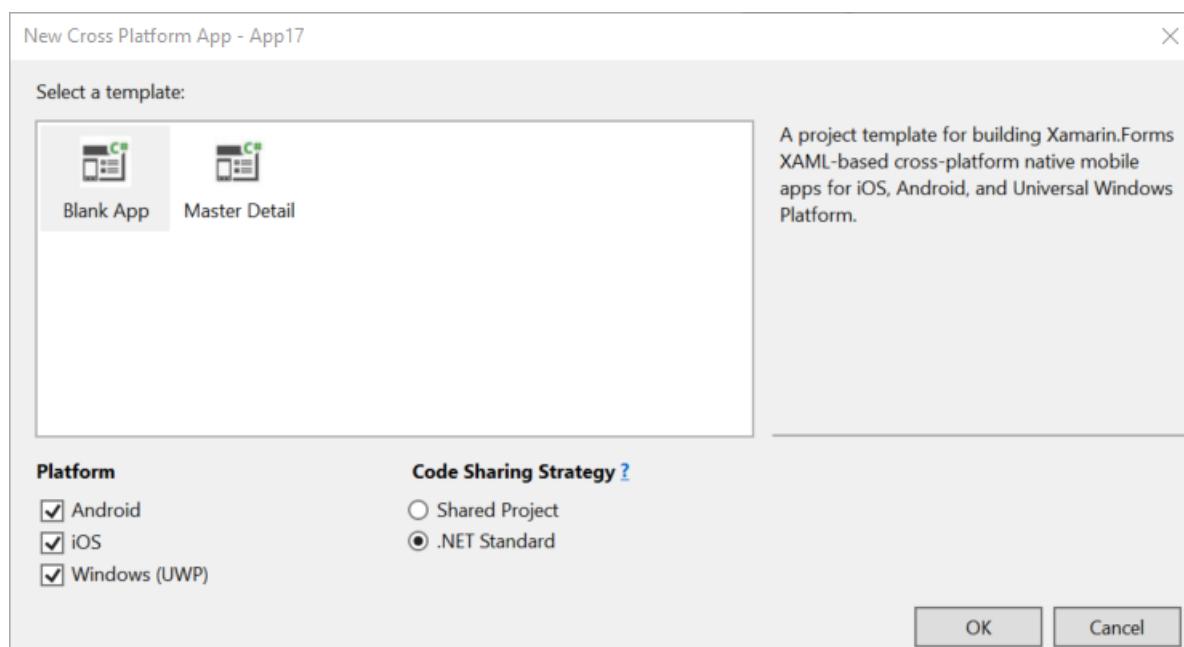
- [Visual Studio](#)
- [Visual Studio for Mac](#)

In Windows, use Visual Studio to select **File > New > Project** from the menu. In the **New Project** dialog, select **Visual C# > Cross Platform** at the left, and then **Mobile App (Xamarin.Forms)** from the list in the center.



Select a location for the solution, give it a name of **XamISamples** (or whatever you prefer), and press **OK**.

On the next screen, select the **Blank App** template and the **.NET Standard** code-sharing strategy:



Press **OK**.

Four projects are created in the solution: the **XamlSamples** .NET Standard library, **XamlSamples.Android**, **XamlSamples.iOS**, and the Universal Windows Platform solution, **XamlSamples.UWP**.

After creating the **XamlSamples** solution, you might want to test your development environment by selecting the various platform projects as the solution startup project, and building and deploying the simple application created by the project template on either phone emulators or real devices.

Unless you need to write platform-specific code, the shared **XamlSamples** .NET Standard library project is where you'll be spending virtually all of your programming time. These articles will not venture outside of that project.

### Anatomy of a XAML File

Within the **XamlSamples** .NET Standard library are a pair of files with the following names:

- **App.xaml**, the XAML file; and
- **App.xaml.cs**, a C# *code-behind* file associated with the XAML file.

You'll need to click the arrow next to **App.xaml** to see the code-behind file.

Both **App.xaml** and **App.xaml.cs** contribute to a class named **App** that derives from **Application**. Most other classes with XAML files contribute to a class that derives from **ContentPage**; those files use XAML to define the visual contents of an entire page. This is true of the other two files in the **XamlSamples** project:

- **MainPage.xaml**, the XAML file; and
- **MainPage.xaml.cs**, the C# code-behind file.

The **MainPage.xaml** file looks like this (although the formatting might be a little different):

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.MainPage">

    <StackLayout>
        <!-- Place new controls here -->
        <Label Text="Welcome to Xamarin Forms!" 
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </StackLayout>

</ContentPage>
```

The two XML namespace (**xmlns**) declarations refer to URIs, the first seemingly on Xamarin's web site and the second on Microsoft's. Don't bother checking what those URIs point to. There's nothing there. They are simply URIs owned by Xamarin and Microsoft, and they basically function as version identifiers.

The first XML namespace declaration means that tags defined within the XAML file with no prefix refer to classes in Xamarin.Forms, for example **ContentPage**. The second namespace declaration defines a prefix of **x**. This is used for several elements and attributes that are intrinsic to XAML itself and which are supported by other implementations of XAML. However, these elements and attributes are slightly different depending on the year embedded in the URI. Xamarin.Forms supports the 2009 XAML specification, but not all of it.

The **local** namespace declaration allows you to access other classes from the .NET Standard library project.

At the end of that first tag, the **x** prefix is used for an attribute named **Class**. Because the use of this **x** prefix is virtually universal for the XAML namespace, XAML attributes such as **Class** are almost always referred to as **x:Class**.

The `x:Class` attribute specifies a fully qualified .NET class name: the `MainPage` class in the `XamlSamples` namespace. This means that this XAML file defines a new class named `MainPage` in the `XamlSamples` namespace that derives from `ContentPage` —the tag in which the `x:Class` attribute appears.

The `x:Class` attribute can only appear in the root element of a XAML file to define a derived C# class. This is the only new class defined in the XAML file. Everything else that appears in the XAML file is instead simply instantiated from existing classes and initialized.

The `MainPage.xaml.cs` file looks like this (aside from unused `using` directives):

```
using Xamarin.Forms;

namespace XamlSamples
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

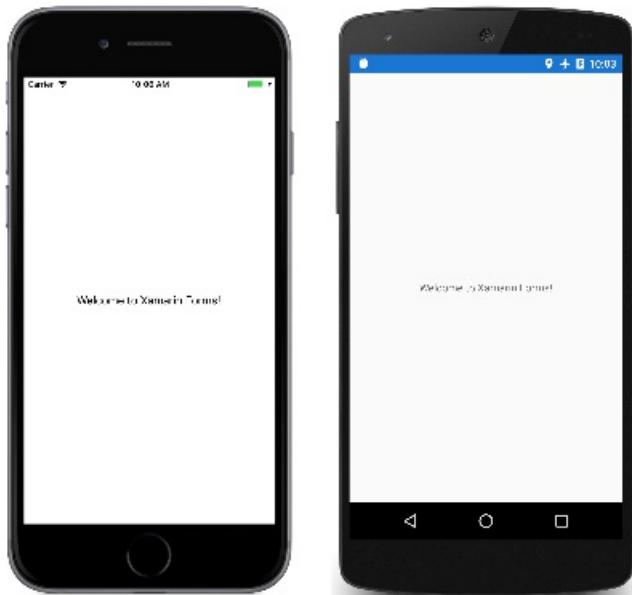
The `MainPage` class derives from `ContentPage`, but notice the `partial` class definition. This suggests that there should be another partial class definition for `MainPage`, but where is it? And what is that `InitializeComponent` method?

When Visual Studio builds the project, it parses the XAML file to generate a C# code file. If you look in the `XamlSamples\XamlSamples\obj\Debug` directory, you'll find a file named `XamlSamples.MainPage.xaml.g.cs`. The 'g' stands for generated. This is the other partial class definition of `MainPage` that contains the definition of the `InitializeComponent` method called from the `MainPage` constructor. These two partial `MainPage` class definitions can then be compiled together. Depending on whether the XAML is compiled or not, either the XAML file or a binary form of the XAML file is embedded in the executable.

At runtime, code in the particular platform project calls a `LoadApplication` method, passing to it a new instance of the `App` class in the .NET Standard library. The `App` class constructor instantiates `MainPage`. The constructor of that class calls `InitializeComponent`, which then calls the `LoadFromXaml` method that extracts the XAML file (or its compiled binary) from the .NET Standard library. `LoadFromXaml` initializes all the objects defined in the XAML file, connects them all together in parent-child relationships, attaches event handlers defined in code to events set in the XAML file, and sets the resultant tree of objects as the content of the page.

Although you normally don't need to spend much time with generated code files, sometimes runtime exceptions are raised on code in the generated files, so you should be familiar with them.

When you compile and run this program, the `Label` element appears in the center of the page as the XAML suggests:

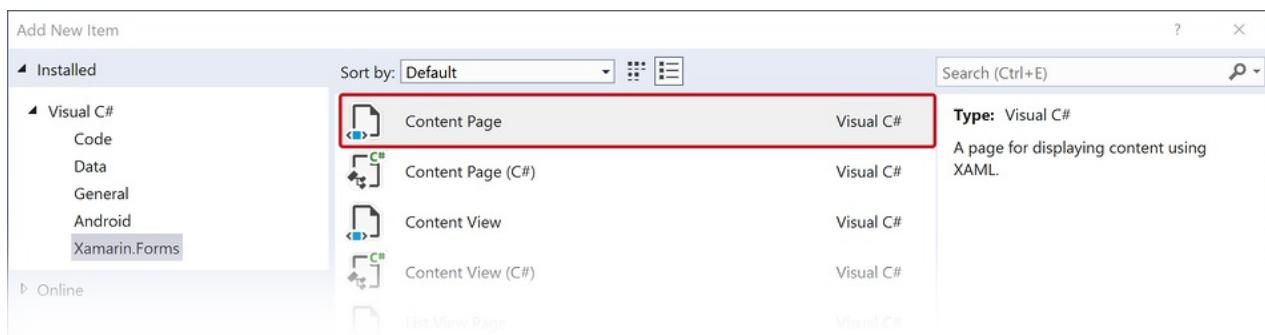


For more interesting visuals, all you need is more interesting XAML.

## Adding New XAML Pages

- [Visual Studio](#)
- [Visual Studio for Mac](#)

To add other XAML-based `ContentPage` classes to your project, select the **XamlSamples** .NET Standard library project and invoke the **Project > Add New Item** menu item. At the left of the **Add New Item** dialog, select **Visual C#** and **Xamarin.Forms**. From the list select **Content Page** (not **Content Page (C#)**), which creates a code-only page, or **Content View**, which is not a page). Give the page a name, for example, **HelloXamlPage.xaml**:



Two files are added to the project, **HelloXamlPage.xaml** and the code-behind file **HelloXamlPage.xaml.cs**.

## Setting Page Content

Edit the **HelloXamlPage.xaml** file so that the only tags are those for `ContentPage` and `ContentPage.Content`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage">
    <ContentPage.Content>
        </ContentPage.Content>
</ContentPage>
```

The `ContentPage.Content` tags are part of the unique syntax of XAML. At first, they might appear to be invalid XML, but they are legal. The period is not a special character in XML.

The `ContentPage.Content` tags are called *property element* tags. `Content` is a property of `ContentPage`, and is generally set to a single view or a layout with child views. Normally properties become attributes in XAML, but it would be hard to set a `Content` attribute to a complex object. For that reason, the property is expressed as an XML element consisting of the class name and the property name separated by a period. Now the `Content` property can be set between the `ContentPage.Content` tags, like this:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.HelloXamlPage"
    Title="Hello XAML Page">
<ContentPage.Content>

    <Label Text="Hello, XAML!"
        VerticalOptions="Center"
        HorizontalTextAlignment="Center"
        Rotation="-15"
        IsVisible="true"
        FontSize="Large"
        FontAttributes="Bold"
        TextColor="Blue" />

</ContentPage.Content>
</ContentPage>
```

Also notice that a `Title` attribute has been set on the root tag.

At this time, the relationship between classes, properties, and XML should be evident: A `Xamarin.Forms` class (such as `ContentPage` or `Label`) appears in the XAML file as an XML element. Properties of that class—including `Title` on `ContentPage` and seven properties of `Label`—usually appear as XML attributes.

Many shortcuts exist to set the values of these properties. Some properties are basic data types: For example, the `Title` and `Text` properties are of type `String`, `Rotation` is of type `Double`, and `IsVisible` (which is `true` by default and is set here only for illustration) is of type `Boolean`.

The `HorizontalTextAlignment` property is of type `TextAlignment`, which is an enumeration. For a property of any enumeration type, all you need supply is a member name.

For properties of more complex types, however, converters are used for parsing the XAML. These are classes in `Xamarin.Forms` that derive from `TypeConverter`. Many are public classes but some are not. For this particular XAML file, several of these classes play a role behind the scenes:

- `LayoutOptionsConverter` for the `VerticalOptions` property
- `FontSizeConverter` for the `FontSize` property
- `ColorTypeConverter` for the `TextColor` property

These converters govern the allowable syntax of the property settings.

The `ThicknessTypeConverter` can handle one, two, or four numbers separated by commas. If one number is supplied, it applies to all four sides. With two numbers, the first is left and right padding, and the second is top and bottom. Four numbers are in the order left, top, right, and bottom.

The `LayoutOptionsConverter` can convert the names of public static fields of the `LayoutOptions` structure to values of type `LayoutOptions`.

The `FontSizeConverter` can handle a `NamedSize` member or a numeric font size.

The `colorTypeConverter` accepts the names of public static fields of the `Color` structure or hexadecimal RGB values, with or without an alpha channel, preceded by a number sign (#). Here's the syntax without an alpha channel:

```
TextColor="#rrggbb"
```

Each of the little letters is a hexadecimal digit. Here is how an alpha channel is included:

```
TextColor="#aarrggbb">
```

For the alpha channel, keep in mind that FF is fully opaque and 00 is fully transparent.

Two other formats allow you to specify only a single hexadecimal digit for each channel:

```
TextColor="#rgb"  TextColor="#argb"
```

In these cases, the digit is repeated to form the value. For example, #CF3 is the RGB color CC-FF-33.

## Page Navigation

When you run the **XamlSamples** program, the `MainPage` is displayed. To see the new `HelloXamlPage` you can either set that as the new startup page in the `App.xaml.cs` file, or navigate to the new page from `MainPage`.

To implement navigation, first change code in the `App.xaml.cs` constructor so that a `NavigationPage` object is created:

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new MainPage());
}
```

In the `MainPage.xaml.cs` constructor, you can create a simple `Button` and use the event handler to navigate to `HelloXamlPage`:

```
public MainPage()
{
    InitializeComponent();

    Button button = new Button
    {
        Text = "Navigate!",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    };

    button.Clicked += async (sender, args) =>
    {
        await Navigation.PushAsync(new HelloXamlPage());
    };

    Content = button;
}
```

Setting the `Content` property of the page replaces the setting of the `Content` property in the XAML file. When you compile and deploy the new version of this program, a button appears on the screen. Pressing it navigates to `HelloXamlPage`. Here's the resultant page on iPhone, Android, and UWP:



You can navigate back to `MainPage` using the < Back button on iOS, using the left arrow at the top of the page or at the bottom of the phone on Android, or using the left arrow at the top of the page on Windows 10.

Feel free to experiment with the XAML for different ways to render the `Label`. If you need to embed any Unicode characters into the text, you can use the standard XML syntax. For example, to put the greeting in smart quotes, use:

```
<Label Text="&#x201C;Hello, XAML!&#x201D;" ... />
```

Here's what it looks like:



## XAML and Code Interactions

The `HelloXamlPage` sample contains only a single `Label` on the page, but this is very unusual. Most `ContentPage` derivatives set the `Content` property to a layout of some sort, such as a `StackLayout`. The `Children` property of the `StackLayout` is defined to be of type `IList<View>` but it's actually an object of type `ElementCollection<View>`, and that collection can be populated with multiple views or other layouts. In XAML, these parent-child relationships are established with normal XML hierarchy. Here's a XAML file for a new page named `XamlPlusCodePage`:

```

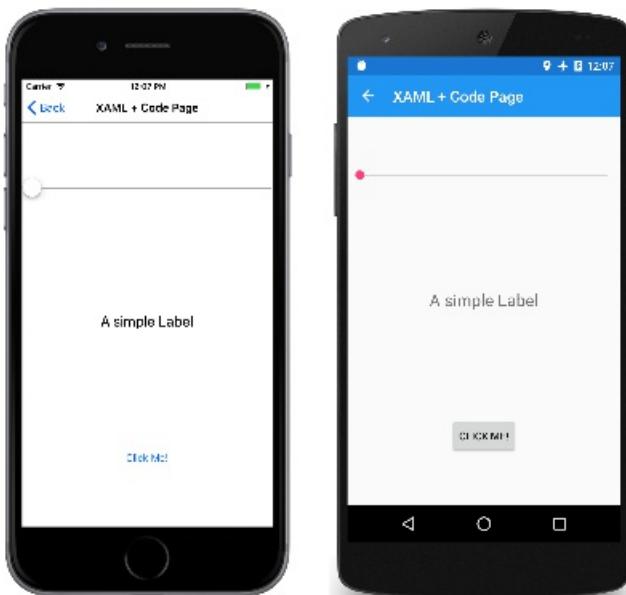
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="CenterAndExpand" />

        <Label Text="A simple Label"
            Font="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

This XAML file is syntactically complete, and here's what it looks like:



However, you are likely to consider this program to be functionally deficient. Perhaps the `Slider` is supposed to cause the `Label` to display the current value, and the `Button` is probably intended to do something within the program.

As you'll see in [Part 4. Data Binding Basics](#), the job of displaying a `Slider` value using a `Label` can be handled entirely in XAML with a data binding. But it is useful to see the code solution first. Even so, handling the `Button` click definitely requires code. This means that the code-behind file for `XamlPlusCodePage` must contain handlers for the `ValueChanged` event of the `Slider` and the `Clicked` event of the `Button`. Let's add them:

```

namespace XamlSamples
{
    public partial class XamlPlusCodePage
    {
        public XamlPlusCodePage()
        {
            InitializeComponent();
        }

        void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
        {

        }

        void OnButtonClicked(object sender, EventArgs args)
        {

        }
    }
}

```

These event handlers do not need to be public.

Back in the XAML file, the `Slider` and `Button` tags need to include attributes for the `ValueChanged` and `Clicked` events that reference these handlers:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="CenterAndExpand"
            ValueChanged="OnSliderValueChanged" />

        <Label Text="A simple Label"
            Font="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Clicked="OnButtonClicked" />
    </StackLayout>
</ContentPage>

```

Notice that assigning a handler to an event has the same syntax as assigning a value to a property.

If the handler for the `ValueChanged` event of the `Slider` will be using the `Label` to display the current value, the handler needs to reference that object from code. The `Label` needs a name, which is specified with the `x:Name` attribute.

```

<Label x:Name="valueLabel"
    Text="A simple Label"
    Font="Large"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand" />

```

The `x` prefix of the `x:Name` attribute indicates that this attribute is intrinsic to XAML.

The name you assign to the `x:Name` attribute has the same rules as C# variable names. For example, it must begin

with a letter or underscore and contain no embedded spaces.

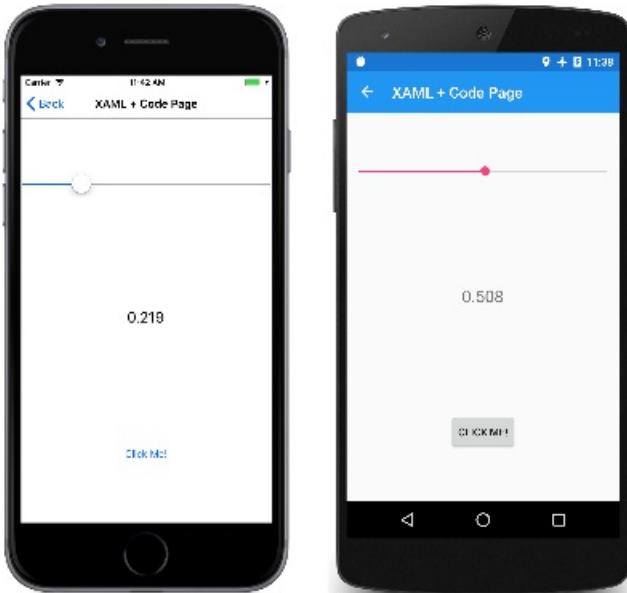
Now the `ValueChanged` event handler can set the `Label` to display the new `Slider` value. The new value is available from the event arguments:

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    valueLabel.Text = args.NewValue.ToString("F3");
}
```

Or, the handler could obtain the `Slider` object that is generating this event from the `sender` argument and obtain the `Value` property from that:

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    valueLabel.Text = ((Slider)sender).Value.ToString("F3");
}
```

When you first run the program, the `Label` doesn't display the `Slider` value because the `ValueChanged` event hasn't yet fired. But any manipulation of the `Slider` causes the value to be displayed:



Now for the `Button`. Let's simulate a response to a `Clicked` event by displaying an alert with the `Text` of the button. The event handler can safely cast the `sender` argument to a `Button` and then access its properties:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
    await DisplayAlert("Clicked!",
        "The button labeled '" + button.Text + "' has been clicked",
        "OK");
}
```

The method is defined as `async` because the `DisplayAlert` method is asynchronous and should be prefaced with the `await` operator, which returns when the method completes. Because this method obtains the `Button` firing the event from the `sender` argument, the same handler could be used for multiple buttons.

You've seen that an object defined in XAML can fire an event that is handled in the code-behind file, and that the code-behind file can access an object defined in XAML using the name assigned to it with the `x:Name` attribute.

These are the two fundamental ways that code and XAML interact.

Some additional insights into how XAML works can be gleaned by examining the newly generated **XamlPlusCode.xaml.g.cs file**, which now includes any name assigned to any `x:Name` attribute as a private field. Here's a simplified version of that file:

```
public partial class XamlPlusCodePage : ContentPage {  
  
    private Label valueLabel;  
  
    private void InitializeComponent() {  
        this.LoadFromXaml(typeof(XamlPlusCodePage));  
        valueLabel = this.FindByName<Label>("valueLabel");  
    }  
}
```

The declaration of this field allows the variable to be freely used anywhere within the `XamlPlusCodePage` partial class file under your jurisdiction. At runtime, the field is assigned after the XAML has been parsed. This means that the `valueLabel` field is `null` when the `XamlPlusCodePage` constructor begins but valid after `InitializeComponent` is called.

After `InitializeComponent` returns control back to the constructor, the visuals of the page have been constructed just as if they had been instantiated and initialized in code. The XAML file no longer plays any role in the class. You can manipulate these objects on the page in any way that you want, for example, by adding views to the `StackLayout`, or setting the `Content` property of the page to something else entirely. You can “walk the tree” by examining the `Content` property of the page and the items in the `Children` collections of layouts. You can set properties on views accessed in this way, or assign event handlers to them dynamically.

Feel free. It’s your page, and XAML is only a tool to build its content.

## Summary

With this introduction, you’ve seen how a XAML file and code file contribute to a class definition, and how the XAML and code files interact. But XAML also has its own unique syntactical features that allow it to be used in a very flexible manner. You can begin exploring these in [Part 2. Essential XAML Syntax](#).

## Related Links

- [XamlSamples](#)
- [Part 2. Essential XAML Syntax](#)
- [Part 3. XAML Markup Extensions](#)
- [Part 4. Data Binding Basics](#)
- [Part 5. From Data Binding to MVVM](#)

# Part 2. Essential XAML Syntax

11/20/2018 • 9 minutes to read • [Edit Online](#)

XAML is mostly designed for instantiating and initializing objects. But often, properties must be set to complex objects that cannot easily be represented as XML strings, and sometimes properties defined by one class must be set on a child class. These two needs require the essential XAML syntax features of property elements and attached properties.

## Property Elements

In XAML, properties of classes are normally set as XML attributes:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="Large"  
      TextColor="Aqua" />
```

However, there is an alternative way to set a property in XAML. To try this alternative with `TextColor`, first delete the existing `TextColor` setting:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="Large" />
```

Open up the empty-element `Label` tag by separating it into start and end tags:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="Large">  
  
</Label>
```

Within these tags, add start and end tags that consist of the class name and a property name separated by a period:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="Large">  
  <Label.TextColor>  
    </Label.TextColor>  
</Label>
```

Set the property value as content of these new tags, like this:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="Large">  
  <Label.TextColor>  
    Aqua  
  </Label.TextColor>  
</Label>
```

These two ways to specify the `TextColor` property are functionally equivalent, but don't use the two ways for the same property because that would effectively be setting the property twice, and might be ambiguous.

With this new syntax, some handy terminology can be introduced:

- `Label` is an *object element*. It is a Xamarin.Forms object expressed as an XML element.
- `Text`, `VerticalOptions`, `FontAttributes` and `FontSize` are *property attributes*. They are Xamarin.Forms properties expressed as XML attributes.
- In that final snippet, `TextColor` has become a *property element*. It is a Xamarin.Forms property but it is now an XML element.

The definition of property elements might at first seem to be a violation of XML syntax, but it's not. The period has no special meaning in XML. To an XML decoder, `Label.TextColor` is simply a normal child element.

In XAML, however, this syntax is very special. One of the rules for property elements is that nothing else can appear in the `Label.TextColor` tag. The value of the property is always defined as content between the property-element start and end tags.

You can use property-element syntax on more than one property:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center">  
  <Label.FontAttributes>  
    Bold  
  </Label.FontAttributes>  
  <Label.FontSize>  
    Large  
  </Label.FontSize>  
  <Label.TextColor>  
    Aqua  
  </Label.TextColor>  
</Label>
```

Or you can use property-element syntax for all the properties:

```

<Label>
    <Label.Text>
        Hello, XAML!
    </Label.Text>
    <Label.FontAttributes>
        Bold
    </Label.FontAttributes>
    <Label.FontSize>
        Large
    </Label.FontSize>
    <Label.TextColor>
        Aqua
    </Label.TextColor>
    <Label.VerticalOptions>
        Center
    </Label.VerticalOptions>
</Label>

```

At first, property-element syntax might seem like an unnecessary long-winded replacement for something comparatively quite simple, and in these examples that is certainly the case.

However, property-element syntax becomes essential when the value of a property is too complex to be expressed as a simple string. Within the property-element tags you can instantiate another object and set its properties. For example, you can explicitly set a property such as `VerticalOptions` to a `LayoutOptions` value with property settings:

```

<Label>
    ...
    <Label.VerticalOptions>
        <LayoutOptions Alignment="Center" />
    </Label.VerticalOptions>
</Label>

```

Another example: The `Grid` has two properties named `RowDefinitions` and `ColumnDefinitions`. These two properties are of type `RowDefinitionCollection` and `ColumnDefinitionCollection`, which are collections of `RowDefinition` and `ColumnDefinition` objects. You need to use property element syntax to set these collections.

Here's the beginning of the XAML file for a `GridDemoPage` class, showing the property element tags for the `RowDefinitions` and `ColumnDefinitions` collections:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.GridDemoPage"
    Title="Grid Demo Page">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>
        ...
    </Grid>
</ContentPage>

```

Notice the abbreviated syntax for defining auto-sized cells, cells of pixel widths and heights, and star settings.

## Attached Properties

You've just seen that the `Grid` requires property elements for the `RowDefinitions` and `ColumnDefinitions` collections to define the rows and columns. However, there must also be some way for the programmer to indicate the row and column where each child of the `Grid` resides.

Within the tag for each child of the `Grid` you specify the row and column of that child using the following attributes:

- `Grid.Row`
- `Grid.Column`

The default values of these attributes are 0. You can also indicate if a child spans more than one row or column with these attributes:

- `Grid.RowSpan`
- `Grid.ColumnSpan`

These two attributes have default values of 1.

Here's the complete GridDemoPage.xaml file:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.GridDemoPage"
    Title="Grid Demo Page">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>

        <Label Text="Autosized cell"
            Grid.Row="0" Grid.Column="0"
            TextColor="White"
            BackgroundColor="Blue" />

        <BoxView Color="Silver"
            HeightRequest="0"
            Grid.Row="0" Grid.Column="1" />

        <BoxView Color="Teal"
            Grid.Row="1" Grid.Column="0" />

        <Label Text="Leftover space"
            Grid.Row="1" Grid.Column="1"
            TextColor="Purple"
            BackgroundColor="Aqua"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

        <Label Text="Span two rows (or more if you want)"
            Grid.Row="0" Grid.Column="2" Grid.RowSpan="2"
            TextColor="Yellow"
            BackgroundColor="Blue"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

        <Label Text="Span two columns"
            Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
            TextColor="Blue"
            BackgroundColor="Yellow"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

        <Label Text="Fixed 100x100"
            Grid.Row="2" Grid.Column="2"
            TextColor="Aqua"
            BackgroundColor="Red"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />

    </Grid>
</ContentPage>

```

The `Grid.Row` and `Grid.Column` settings of 0 are not required but are generally included for purposes of clarity.

Here's what it looks like:



Judging solely from the syntax, these `Grid.Row`, `Grid.Column`, `Grid.RowStyle`, and `Grid.ColumnSpan` attributes appear to be static fields or properties of `Grid`, but interestingly enough, `Grid` does not define anything named `Row`, `Column`, `RowSpan`, or `ColumnSpan`.

Instead, `Grid` defines four bindable properties named `RowProperty`, `ColumnProperty`, `RowSpanProperty`, and `ColumnSpanProperty`. These are special types of bindable properties known as *attached properties*. They are defined by the `Grid` class but set on children of the `Grid`.

When you wish to use these attached properties in code, the `Grid` class provides static methods named `SetRow`, `GetColumn`, and so forth. But in XAML, these attached properties are set as attributes in the children of the `Grid` using simple properties names.

Attached properties are always recognizable in XAML files as attributes containing both a class and a property name separated by a period. They are called *attached properties* because they are defined by one class (in this case, `Grid`) but attached to other objects (in this case, children of the `Grid`). During layout, the `Grid` can interrogate the values of these attached properties to know where to place each child.

The `AbsoluteLayout` class defines two attached properties named `LayoutBounds` and `LayoutFlags`. Here's a checkerboard pattern realized using the proportional positioning and sizing features of `AbsoluteLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.AbsoluteDemoPage"
    Title="Absolute Demo Page">

    <AbsoluteLayout BackgroundColor="#FF8080">
        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0.33, 0, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="1, 0, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0, 0.33, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0.67, 0.33, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0.33, 0.67, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

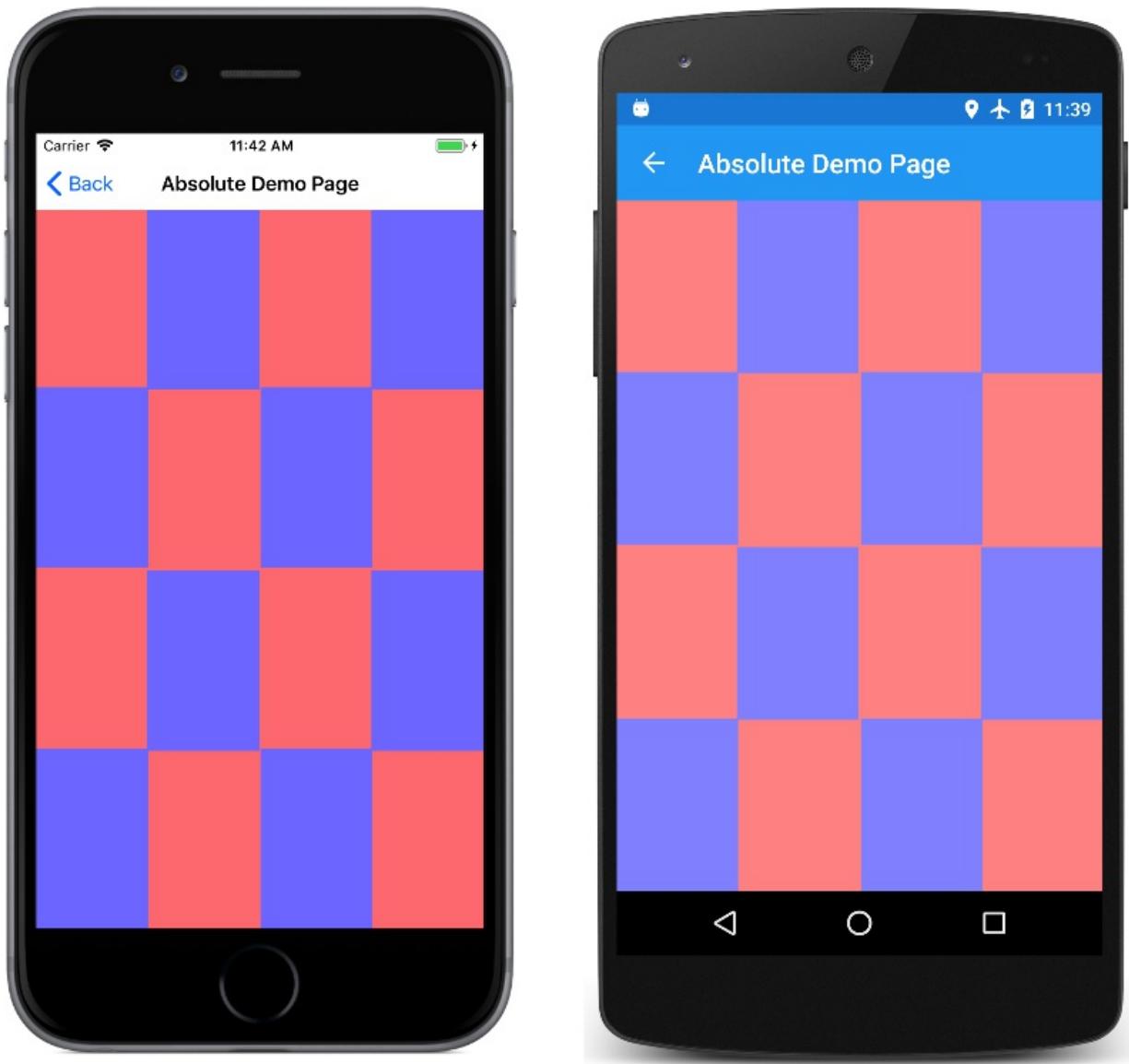
        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="1, 0.67, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0, 1, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0.67, 1, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

    </AbsoluteLayout>
</ContentPage>
```

And here it is:



For something like this, you might question the wisdom of using XAML. Certainly, the repetition and regularity of the `LayoutBounds` rectangle suggests that it might be better realized in code.

That's certainly a legitimate concern, and there's no problem with balancing the use of code and markup when defining your user interfaces. It's easy to define some of the visuals in XAML and then use the constructor of the code-behind file to add some more visuals that might be better generated in loops.

## Content Properties

In the previous examples, the `StackLayout`, `Grid`, and `AbsoluteLayout` objects are set to the `Content` property of the `ContentPage`, and the children of these layouts are actually items in the `children` collection. Yet these `Content` and `children` properties are nowhere in the XAML file.

You can certainly include the `Content` and `children` properties as property elements, such as in the **XamlPlusCode** sample:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout.Children>
                <Slider VerticalOptions="CenterAndExpand"
                    ValueChanged="OnSliderValueChanged" />

                <Label x:Name="valueLabel"
                    Text="A simple Label"
                    FontSize="Large"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand" />

                <Button Text="Click Me!"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand"
                    Clicked="OnButtonClicked" />
            </StackLayout.Children>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

The real question is: Why are these property elements *not* required in the XAML file?

Elements defined in Xamarin.Forms for use in XAML are allowed to have one property flagged in the `ContentProperty` attribute on the class. If you look up the `ContentPage` class in the online Xamarin.Forms documentation, you'll see this attribute:

```
[Xamarin.Forms.ContentProperty("Content")]
public class ContentPage : TemplatedPage
```

This means that the `Content` property-element tags are not required. Any XML content that appears between the start and end `ContentPage` tags is assumed to be assigned to the `content` property.

`StackLayout`, `Grid`, `AbsoluteLayout`, and `RelativeLayout` all derive from `Layout<View>`, and if you look up `Layout<T>` in the Xamarin.Forms documentation, you'll see another `ContentProperty` attribute:

```
[Xamarin.Forms.ContentProperty("Children")]
public abstract class Layout<T> : Layout ...
```

That allows content of the layout to be automatically added to the `Children` collection without explicit `Children` property-element tags.

Other classes also have `ContentProperty` attribute definitions. For example, the content property of `Label` is `Text`. Check the API documentation for others.

## Platform Differences with OnPlatform

In single page applications, it is common to set the `Padding` property on the page to avoid overwriting the iOS status bar. In code, you can use the `Device.RuntimePlatform` property for this purpose:

```
if (Device.RuntimePlatform == Device.iOS)
{
    Padding = new Thickness(0, 20, 0, 0);
}
```

You can also do something similar in XAML using the `OnPlatform` and `On` classes. First include property elements for the `Padding` property near the top of the page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        ...
    </ContentPage.Padding>
</ContentPage>
```

Within these tags, include an `OnPlatform` tag. `OnPlatform` is a generic class. You need to specify the generic type argument, in this case, `Thickness`, which is the type of `Padding` property. Fortunately, there's a XAML attribute specifically to define generic arguments called `x>TypeArguments`. This should match the type of the property you're setting:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x>TypeArguments="Thickness">
            ...
        </OnPlatform>
    </ContentPage.Padding>
...
</ContentPage>
```

`OnPlatform` has a property named `Platforms` that is an `IList` of `On` objects. Use property element tags for that property:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x>TypeArguments="Thickness">
            <OnPlatform.Platforms>
                ...
            </OnPlatform.Platforms>
        </OnPlatform>
    </ContentPage.Padding>
...
</ContentPage>
```

Now add `On` elements. For each one set the `Platform` property and the `Value` property to markup for the `Thickness` property:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <OnPlatform.Platforms>
                <On Platform="iOS" Value="0, 20, 0, 0" />
                <On Platform="Android" Value="0, 0, 0, 0" />
                <On Platform="UWP" Value="0, 0, 0, 0" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>

```

This markup can be simplified. The content property of `OnPlatform` is `Platforms`, so those property-element tags can be removed:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
            <On Platform="Android" Value="0, 0, 0, 0" />
            <On Platform="UWP" Value="0, 0, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>

```

The `Platform` property of `On` is of type `IList<string>`, so you can include multiple platforms if the values are the same:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
            <On Platform="Android, UWP" Value="0, 0, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>

```

Because Android and UWP are set to the default value of `Padding`, that tag can be removed:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>
```

This is the standard way to set a platform-dependent `Padding` property in XAML. If the `Value` setting cannot be represented by a single string, you can define property elements for it:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS">
                <On.Value>
                    0, 20, 0, 0
                </On.Value>
            </On>
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>
```

## Summary

With property elements and attached properties, much of the basic XAML syntax has been established. However, sometimes you need to set properties to objects in an indirect manner, for example, from a resource dictionary. This approach is covered in the next part, Part 3. [XAML Markup Extensions](#).

## Related Links

- [XamlSamples](#)
- [Part 1. Getting Started with XAML](#)
- [Part 3. XAML Markup Extensions](#)
- [Part 4. Data Binding Basics](#)
- [Part 5. From Data Binding to MVVM](#)

# Part 3. XAML Markup Extensions

11/12/2018 • 11 minutes to read • [Edit Online](#)

*XAML markup extensions constitute an important feature in XAML that allow properties to be set to objects or values that are referenced indirectly from other sources. XAML markup extensions are particularly important for sharing objects, and referencing constants used throughout an application, but they find their greatest utility in data bindings.*

## XAML Markup Extensions

In general, you use XAML to set properties of an object to explicit values, such as a string, a number, an enumeration member, or a string that is converted to a value behind the scenes.

Sometimes, however, properties must instead reference values defined somewhere else, or which might require a little processing by code at runtime. For these purposes, XAML *markup extensions* are available.

These XAML markup extensions are not extensions of XML. XAML is entirely legal XML. They're called "extensions" because they are backed by code in classes that implement `IMarkupExtension`. You can write your own custom markup extensions.

In many cases, XAML markup extensions are instantly recognizable in XAML files because they appear as attribute settings delimited by curly braces: { and }, but sometimes markup extensions appear in markup as conventional elements.

## Shared Resources

Some XAML pages contain several views with properties set to the same values. For example, many of the property settings for these `Button` objects are the same:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">

    <StackLayout>
        <Button Text="Do this!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            FontSize="24" />

        <Button Text="Do that!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            FontSize="24" />

        <Button Text="Do the other thing!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            Rotation="-15"
            TextColor="Red"
            FontSize="24" />

    </StackLayout>
</ContentPage>

```

If one of these properties needs to be changed, you might prefer to make the change just once rather than three times. If this were code, you'd likely be using constants and static read-only objects to help keep such values consistent and easy to modify.

In XAML, one popular solution is to store such values or objects in a *resource dictionary*. The `visualElement` class defines a property named `Resources` of type `ResourceDictionary`, which is a dictionary with keys of type `string` and values of type `object`. You can put objects into this dictionary and then reference them from markup, all in XAML.

To use a resource dictionary on a page, include a pair of `Resources` property-element tags. It's most convenient to put these at the top of the page:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">

    <ContentPage.Resources>
        ...
    </ContentPage.Resources>
</ContentPage>

```

It's also necessary to explicitly include `ResourceDictionary` tags:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>

            </ResourceDictionary>
        </ContentPage.Resources>
        ...
    </ContentPage>

```

Now objects and values of various types can be added to the resource dictionary. These types must be instantiable. They can't be abstract classes, for example. These types must also have a public parameterless constructor. Each item requires a dictionary key specified with the `x:Key` attribute. For example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions"
                Alignment="Center" />

            <LayoutOptions x:Key="vertOptions"
                Alignment="Center"
                Expands="True" />
        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>

```

These two items are values of the structure type `LayoutOptions`, and each has a unique key and one or two properties set. In code and markup, it's much more common to use the static fields of `LayoutOptions`, but here it's more convenient to set the properties.

Now it's necessary to set the `HorizontalOptions` and `VerticalOptions` properties of these buttons to these resources, and that's done with the `StaticResource` XAML markup extension:

```

<Button Text="Do this!" 
    HorizontalOptions="{StaticResource horzOptions}"
    VerticalOptions="{StaticResource vertOptions}"
    BorderWidth="3"
    Rotation="-15"
    TextColor="Red"
    FontSize="24" />

```

The `StaticResource` markup extension is always delimited with curly braces, and includes the dictionary key.

The name `StaticResource` distinguishes it from `DynamicResource`, which Xamarin.Forms also supports. `DynamicResource` is for dictionary keys associated with values that might change during runtime, while `StaticResource` accesses elements from the dictionary just once when the elements on the page are constructed.

For the `BorderWidth` property, it's necessary to store a double in the dictionary. XAML conveniently defines tags for common data types like `x:Double` and `x:Int32`:

```

<ContentPage.Resources>
    <ResourceDictionary>
        <LayoutOptions x:Key="horzOptions"
            Alignment="Center" />

        <LayoutOptions x:Key="vertOptions"
            Alignment="Center"
            Expands="True" />

        <x:Double x:Key="borderWidth">
            3
        </x:Double>
    </ResourceDictionary>
</ContentPage.Resources>

```

You don't need to put it on three lines. This dictionary entry for this rotation angle only takes up one line:

```

<ContentPage.Resources>
    <ResourceDictionary>
        <LayoutOptions x:Key="horzOptions"
            Alignment="Center" />

        <LayoutOptions x:Key="vertOptions"
            Alignment="Center"
            Expands="True" />

        <x:Double x:Key="borderWidth">
            3
        </x:Double>

        <x:Double x:Key="rotationAngle">-15</x:Double>
    </ResourceDictionary>
</ContentPage.Resources>

```

Those two resources can be referenced in the same way as the `LayoutOptions` values:

```

<Button Text="Do this!">
    HorizontalOptions="{StaticResource horzOptions}"
    VerticalOptions="{StaticResource vertOptions}"
    BorderWidth="{StaticResource borderWidth}"
    Rotation="{StaticResource rotationAngle}"
    TextColor="Red"
    FontSize="24" />

```

For resources of type `Color`, you can use the same string representations that you use when directly assigning attributes of these types. The type converters are invoked when the resource is created. Here's a resource of type `Color`:

```
<Color x:Key="textColor">Red</Color>
```

Often, programs set a `FontSize` property to a member of the `NamedSize` enumeration such as `Large`. The `FontSizeConverter` class works behind the scenes to convert it into a platform-dependent value using the `Device.GetNamedSized` method. However, when defining a font-size resource, it makes more sense to use a numeric value, shown here as an `x:Double` type:

```
<x:Double x:Key="fontSize">24</x:Double>
```

Now all the properties except `Text` are defined by resource settings:

```
<Button Text="Do this!"  
       HorizontalOptions="{StaticResource horzOptions}"  
       VerticalOptions="{StaticResource vertOptions}"  
       BorderWidth="{StaticResource borderWidth}"  
       Rotation="{StaticResource rotationAngle}"  
       TextColor="{StaticResource textColor}"  
       FontSize="{StaticResource fontSize}" />
```

It's also possible to use `OnPlatform` within the resource dictionary to define different values for the platforms.

Here's how an `OnPlatform` object can be part of the resource dictionary for different text colors:

```
<OnPlatform x:Key="textColor"  
            x:TypeArguments="Color">  
    <On Platform="iOS" Value="Red" />  
    <On Platform="Android" Value="Aqua" />  
    <On Platform="UWP" Value="#80FF80" />  
</OnPlatform>
```

Notice that `OnPlatform` gets both an `x:Key` attribute because it's an object in the dictionary and an `x:TypeArguments` attribute because it's a generic class. The `iOS`, `Android`, and `UWP` attributes are converted to `Color` values when the object is initialized.

Here's the final complete XAML file with three buttons accessing six shared values:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions"
                Alignment="Center" />

            <LayoutOptions x:Key="vertOptions"
                Alignment="Center"
                Expands="True" />

            <x:Double x:Key="borderWidth">3</x:Double>

            <x:Double x:Key="rotationAngle">-15</x:Double>

            <OnPlatform x:Key="textColor"
                x:TypeArguments="Color">
                <On Platform="iOS" Value="Red" />
                <On Platform="Android" Value="Aqua" />
                <On Platform="UWP" Value="#80FF80" />
            </OnPlatform>

            <x:String x:Key="fontSize">Large</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>

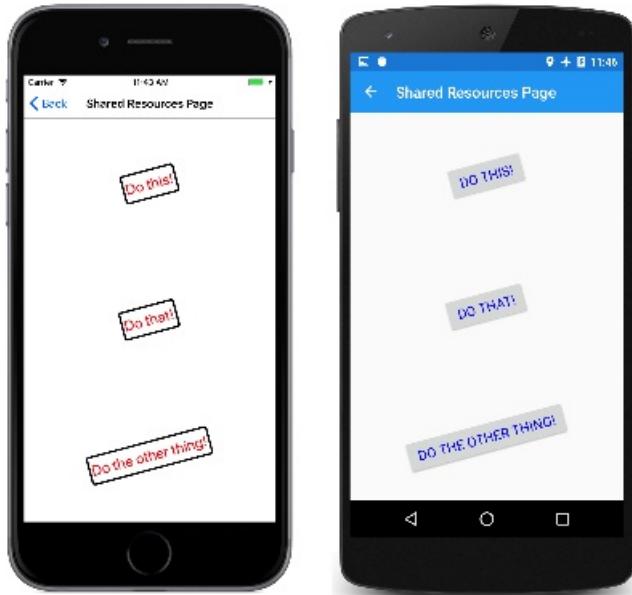
    <StackLayout>
        <Button Text="Do this!">
            HorizontalOptions="{StaticResource horzOptions}"
            VerticalOptions="{StaticResource vertOptions}"
            BorderWidth="{StaticResource borderWidth}"
            Rotation="{StaticResource rotationAngle}"
            TextColor="{StaticResource textColor}"
            FontSize="{StaticResource fontSize}" />

        <Button Text="Do that!">
            HorizontalOptions="{StaticResource horzOptions}"
            VerticalOptions="{StaticResource vertOptions}"
            BorderWidth="{StaticResource borderWidth}"
            Rotation="{StaticResource rotationAngle}"
            TextColor="{StaticResource textColor}"
            FontSize="{StaticResource fontSize}" />

        <Button Text="Do the other thing!">
            HorizontalOptions="{StaticResource horzOptions}"
            VerticalOptions="{StaticResource vertOptions}"
            BorderWidth="{StaticResource borderWidth}"
            Rotation="{StaticResource rotationAngle}"
            TextColor="{StaticResource textColor}"
            FontSize="{StaticResource fontSize}" />
    </StackLayout>
</ContentPage>

```

The screenshots verify the consistent styling, and the platform-dependent styling:



Although it is most common to define the `Resources` collection at the top of the page, keep in mind that the `Resources` property is defined by `VisualElement`, and you can have `Resources` collections on other elements on the page. For example, try adding one to the `StackLayout` in this example:

```
<StackLayout>
    <StackLayout.Resources>
        <ResourceDictionary>
            <Color x:Key="textColor">Blue</Color>
        </ResourceDictionary>
    </StackLayout.Resources>
    ...
</StackLayout>
```

You'll discover that the text color of the buttons is now blue. Basically, whenever the XAML parser encounters a `StaticResource` markup extension, it searches up the visual tree and uses the first `ResourceDictionary` it encounters containing that key.

One of the most common types of objects stored in resource dictionaries is the Xamarin.Forms `Style`, which defines a collection of property settings. Styles are discussed in the article [Styles](#).

Sometimes developers new to XAML wonder if they can put a visual element such as `Label` or `Button` in a `ResourceDictionary`. While it's surely possible, it doesn't make much sense. The purpose of the `ResourceDictionary` is to share objects. A visual element cannot be shared. The same instance cannot appear twice on a single page.

## The `x:Static` Markup Extension

Despite the similarities of their names, `x:Static` and `StaticResource` are very different. `StaticResource` returns an object from a resource dictionary while `x:Static` accesses one of the following:

- a public static field
- a public static property
- a public constant field
- an enumeration member.

The `StaticResource` markup extension is supported by XAML implementations that define a resource dictionary, while `x:Static` is an intrinsic part of XAML, as the `x` prefix reveals.

Here are a few examples that demonstrate how `x:Static` can explicitly reference static fields and enumeration members:

```
<Label Text="Hello, XAML!"  
    VerticalOptions="{x:Static LayoutOptions.Start}"  
    HorizontalTextAlignment="{x:Static TextAlignment.Center}"  
    TextColor="{x:Static Color.Aqua}" />
```

So far, this is not very impressive. But the `x:Static` markup extension can also reference static fields or properties from your own code. For example, here's an `AppConstants` class that contains some static fields that you might want to use on multiple pages throughout an application:

```
using System;  
using Xamarin.Forms;  
  
namespace XamlSamples  
{  
    static class AppConstants  
    {  
        public static readonly Thickness PagePadding;  
  
        public static readonly Font TitleFont;  
  
        public static readonly Color BackgroundColor = Color.Aqua;  
  
        public static readonly Color ForegroundColor = Color.Brown;  
  
        static AppConstants()  
        {  
            switch (Device.RuntimePlatform)  
            {  
                case Device.iOS:  
                    PagePadding = new Thickness(5, 20, 5, 0);  
                    TitleFont = Font.SystemFontOfSize(35, FontAttributes.Bold);  
                    break;  
  
                case Device.Android:  
                    PagePadding = new Thickness(5, 0, 5, 0);  
                    TitleFont = Font.SystemFontOfSize(40, FontAttributes.Bold);  
                    break;  
  
                case Device.UWP:  
                    PagePadding = new Thickness(5, 0, 5, 0);  
                    TitleFont = Font.SystemFontOfSize(50, FontAttributes.Bold);  
                    break;  
            }  
        }  
    }  
}
```

To reference the static fields of this class in the XAML file, you'll need some way to indicate within the XAML file where this file is located. You do this with an XML namespace declaration.

Recall that the XAML files created as part of the standard `Xamarin.Forms` XAML template contain two XML namespace declarations: one for accessing `Xamarin.Forms` classes and another for referencing tags and attributes intrinsic to XAML:

```
xmlns="http://xamarin.com/schemas/2014/forms"  
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

You'll need additional XML namespace declarations to access other classes. Each additional XML namespace

declaration defines a new prefix. To access classes local to the shared application .NET Standard library, such as `AppConstants`, XAML programmers often use the prefix `local`. The namespace declaration must indicate the CLR (Common Language Runtime) namespace name, also known as the .NET namespace name, which is the name that appears in a C# `namespace` definition or in a `using` directive:

```
xmlns:local="clr-namespace:XamlSamples"
```

You can also define XML namespace declarations for .NET namespaces in any assembly that the .NET Standard library references. For example, here's a `sys` prefix for the standard .NET `System` namespace, which is in the **mscorlib** assembly, which once stood for "Microsoft Common Object Runtime Library," but now means "Multilanguage Standard Common Object Runtime Library." Because this is another assembly, you must also specify the assembly name, in this case **mscorlib**:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Notice that the keyword `clr-namespace` is followed by a colon and then the .NET namespace name, followed by a semicolon, the keyword `assembly`, an equal sign, and the assembly name.

Yes, a colon follows `clr-namespace` but equal sign follows `assembly`. The syntax was defined in this way deliberately: Most XML namespace declarations reference a URI that begins a URI scheme name such as `http`, which is always followed by a colon. The `clr-namespace` part of this string is intended to mimic that convention.

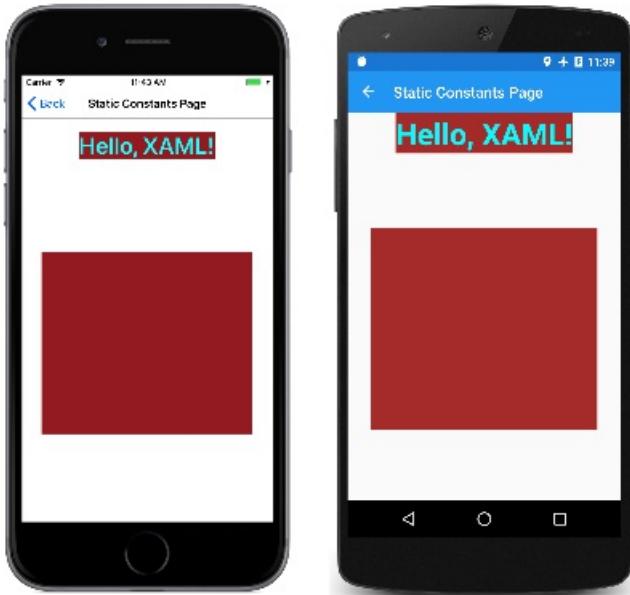
Both these namespace declarations are included in the **StaticConstantsPage** sample. Notice that the `BoxView` dimensions are set to `Math.PI` and `Math.E`, but scaled by a factor of 100:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="XamlSamples.StaticConstantsPage"
    Title="Static Constants Page"
    Padding="{x:Static local:AppConstants.PagePadding}">

    <StackLayout>
        <Label Text="Hello, XAML!">
            TextColor="{x:Static local:AppConstants.BackgroundColor}"
            BackgroundColor="{x:Static local:AppConstants.ForegroundColor}"
            Font="{x:Static local:AppConstants.TitleFont}"
            HorizontalOptions="Center" />

        <BoxView WidthRequest="{x:Static sys:Math.PI}"
            HeightRequest="{x:Static sys:Math.E}"
            Color="{x:Static local:AppConstants.ForegroundColor}"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Scale="100" />
    </StackLayout>
</ContentPage>
```

The size of the resultant `BoxView` relative to the screen is platform-dependent:



## Other Standard Markup Extensions

Several markup extensions are intrinsic to XAML and supported in Xamarin.Forms XAML files. Some of these are not used very often but are essential when you need them:

- If a property has a non- `null` value by default but you want to set it to `null`, set it to the `{x:Null}` markup extension.
- If a property is of type `Type`, you can assign it to a `Type` object using the markup extension `{x:Type someClass}`.
- You can define arrays in XAML using the `x:Array` markup extension. This markup extension has a required attribute named `Type` that indicates the type of the elements in the array.
- The `Binding` markup extension is discussed in [Part 4. Data Binding Basics](#).

## The ConstraintExpression Markup Extension

Markup extensions can have properties, but they are not set like XML attributes. In a markup extension, property settings are separated by commas, and no quotation marks appear within the curly braces.

This can be illustrated with the Xamarin.Forms markup extension named `ConstraintExpression`, which is used with the `RelativeLayout` class. You can specify the location or size of a child view as a constant, or relative to a parent or other named view. The syntax of the `ConstraintExpression` allows you set the position or size of a view using a `Factor` times a property of another view, plus a `Constant`. Anything more complex than that requires code.

Here's an example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.RelativeLayoutPage"
    Title="RelativeLayout Page">

    <RelativeLayout>

        <!-- Upper left -->
        <BoxView Color="Red"
            RelativeLayout.XConstraint=
                "{ConstraintExpression Type=Constant,
                    Constant=0}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=Constant,
                    Constant=0}">
```

```

        "{ConstraintExpression Type=Constant,
                           Constant=0}" />

<!-- Upper right -->
<BoxView Color="Green"
          RelativeLayout.XConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Width,
                                  Factor=1,
                                  Constant=-40}"
          RelativeLayout.YConstraint=
            "{ConstraintExpression Type=Constant,
                                  Constant=0}" />

<!-- Lower left -->
<BoxView Color="Blue"
          RelativeLayout.XConstraint=
            "{ConstraintExpression Type=Constant,
                                  Constant=0}"
          RelativeLayout.YConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Height,
                                  Factor=1,
                                  Constant=-40}" />

<!-- Lower right -->
<BoxView Color="Yellow"
          RelativeLayout.XConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Width,
                                  Factor=1,
                                  Constant=-40}"
          RelativeLayout.YConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Height,
                                  Factor=1,
                                  Constant=-40}" />

<!-- Centered and 1/3 width and height of parent -->
<BoxView x:Name="oneThird"
          Color="Red"
          RelativeLayout.XConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Width,
                                  Factor=0.33}"
          RelativeLayout.YConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Height,
                                  Factor=0.33}"
          RelativeLayout.WidthConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Width,
                                  Factor=0.33}"
          RelativeLayout.HeightConstraint=
            "{ConstraintExpression Type=RelativeToParent,
                                  Property=Height,
                                  Factor=0.33}" />

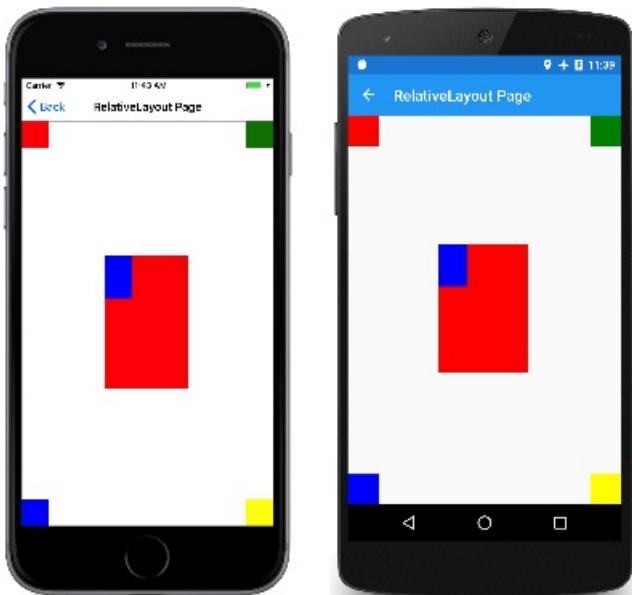
<!-- 1/3 width and height of previous -->
<BoxView Color="Blue"
          RelativeLayout.XConstraint=
            "{ConstraintExpression Type=RelativeToView,
                                  ElementName=oneThird,
                                  Property=X}"
          RelativeLayout.YConstraint=
            "{ConstraintExpression Type=RelativeToView,
                                  ElementName=oneThird,
                                  Property=Y}"
          RelativeLayout.WidthConstraint=
            "{ConstraintExpression Type=RelativeToView,
                                  ElementName=oneThird,
                                  Property=Width,
                                  Factor=0.33}"

```

```
        Factor=0.33}" />
    RelativeLayout.HeightConstraint=
        "{ConstraintExpression Type=RelativeToView,
            ElementName=oneThird,
            Property=Height,
            Factor=0.33}" />
</RelativeLayout>
</ContentPage>
```

Perhaps the most important lesson you should take from this sample is the syntax of the markup extension: No quotation marks must appear within the curly braces of a markup extension. When typing the markup extension in a XAML file, it is natural to want to enclose the values of the properties in quotation marks. Resist the temptation!

Here's the program running:



## Summary

The XAML markup extensions shown here provide important support for XAML files. But perhaps the most valuable XAML markup extension is `Binding`, which is covered in the next part of this series, [Part 4. Data Binding Basics](#).

## Related Links

- [XamlSamples](#)
- [Part 1. Getting Started with XAML](#)
- [Part 2. Essential XAML Syntax](#)
- [Part 4. Data Binding Basics](#)
- [Part 5. From Data Binding to MVVM](#)

# Part 4. Data Binding Basics

11/20/2018 • 11 minutes to read • [Edit Online](#)

*Data bindings allow properties of two objects to be linked so that a change in one causes a change in the other. This is a very valuable tool, and while data bindings can be defined entirely in code, XAML provides shortcuts and convenience. Consequently, one of the most important markup extensions in Xamarin.Forms is Binding.*

## Data Bindings

Data bindings connect properties of two objects, called the *source* and the *target*. In code, two steps are required: The `BindingContext` property of the target object must be set to the source object, and the `SetBinding` method (often used in conjunction with the `Binding` class) must be called on the target object to bind a property of that object to a property of the source object.

The target property must be a bindable property, which means that the target object must derive from `BindableObject`. The online Xamarin.Forms documentation indicates which properties are bindable properties. A property of `Label` such as `Text` is associated with the bindable property `TextProperty`.

In markup, you must also perform the same two steps that are required in code, except that the `Binding` markup extension takes the place of the `SetBinding` call and the `Binding` class.

However, when you define data bindings in XAML, there are multiple ways to set the `BindingContext` of the target object. Sometimes it's set from the code-behind file, sometimes using a `StaticResource` or `x:Static` markup extension, and sometimes as the content of `BindingContext` property-element tags.

Bindings are used most often to connect the visuals of a program with an underlying data model, usually in a realization of the MVVM (Model-View-ViewModel) application architecture, as discussed in [Part 5. From Data Bindings to MVVM](#), but other scenarios are possible.

## View-to-View Bindings

You can define data bindings to link properties of two views on the same page. In this case, you set the `BindingContext` of the target object using the `x:Reference` markup extension.

Here's a XAML file that contains a `Slider` and two `Label` views, one of which is rotated by the `Slider` value and another which displays the `Slider` value:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SliderBindingsPage"
    Title="Slider Bindings Page">

    <StackLayout>
        <Label Text="ROTATION"
            BindingContext="{x:Reference Name=slider}"
            Rotation="{Binding Path=Value}"
            FontAttributes="Bold"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="CenterAndExpand" />

        <Label BindingContext="{x:Reference slider}"
            Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
            FontAttributes="Bold"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

The `Slider` contains an `x:Name` attribute that is referenced by the two `Label` views using the `x:Reference` markup extension.

The `x:Reference` binding extension defines a property named `Name` to set to the name of the referenced element, in this case `slider`. However, the `ReferenceExtension` class that defines the `x:Reference` markup extension also defines a `ContentProperty` attribute for `Name`, which means that it isn't explicitly required. Just for variety, the first `x:Reference` includes "Name=" but the second does not:

```

BindingContext="{x:Reference Name=slider}"
...
BindingContext="{x:Reference slider}"

```

The `Binding` markup extension itself can have several properties, just like the `BindingBase` and `Binding` class. The `ContentProperty` for `Binding` is `Path`, but the "Path=" part of the markup extension can be omitted if the path is the first item in the `Binding` markup extension. The first example has "Path=" but the second example omits it:

```

Rotation="{Binding Path=Value}"
...
Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"

```

The properties can all be on one line or separated into multiple lines:

```

Text="{Binding Value,
    StringFormat='The angle is {0:F0} degrees'}"

```

Do whatever is convenient.

Notice the `StringFormat` property in the second `Binding` markup extension. In Xamarin.Forms, bindings do not perform any implicit type conversions, and if you need to display a non-string object as a string you must

provide a type converter or use `StringFormat`. Behind the scenes, the static `String.Format` method is used to implement `StringFormat`. That's potentially a problem, because .NET formatting specifications involve curly braces, which are also used to delimit markup extensions. This creates a risk of confusing the XAML parser. To avoid that, put the entire formatting string in single quotation marks:

```
Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
```

Here's the running program:



## The Binding Mode

A single view can have data bindings on several of its properties. However, each view can have only one `BindingContext`, so multiple data bindings on that view must all reference properties of the same object.

The solution to this and other problems involves the `Mode` property, which is set to a member of the `BindingMode` enumeration:

- `Default`
- `OneWay` — values are transferred from the source to the target
- `OneWayToSource` — values are transferred from the target to the source
- `TwoWay` — values are transferred both ways between source and target

The following program demonstrates one common use of the `OneWayToSource` and `TwoWay` binding modes. Four `Slider` views are intended to control the `Scale`, `Rotate`, `RotateX`, and `RotateY` properties of a `Label`. At first, it seems as if these four properties of the `Label` should be data-binding targets because each is being set by a `Slider`. However, the `BindingContext` of `Label` can be only one object, and there are four different sliders.

For that reason, all the bindings are set in seemingly backwards ways: The `BindingContext` of each of the four sliders is set to the `Label`, and the bindings are set on the `Value` properties of the sliders. By using the `OneWayToSource` and `TwoWay` modes, these `Value` properties can set the source properties, which are the `Scale`, `Rotate`, `RotateX`, and `RotateY` properties of the `Label`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SliderTransformsPage"
    Padding="5">
    <!-- Content -->

```

```

    Title="Slider Transforms Page">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <!-- Scaled and rotated Label -->
    <Label x:Name="label"
        Text="TEXT"
        HorizontalOptions="Center"
        VerticalOptions="CenterAndExpand" />

    <!-- Slider and identifying Label for Scale -->
    <Slider x:Name="scaleSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="1" Grid.Column="0"
        Maximum="10"
        Value="{Binding Scale, Mode=TwoWay}" />

    <Label BindingContext="{x:Reference scaleSlider}"
        Text="{Binding Value, StringFormat='Scale = {0:F1}'}"
        Grid.Row="1" Grid.Column="1"
        VerticalTextAlignment="Center" />

    <!-- Slider and identifying Label for Rotation -->
    <Slider x:Name="rotationSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="2" Grid.Column="0"
        Maximum="360"
        Value="{Binding Rotation, Mode=OneWayToSource}" />

    <Label BindingContext="{x:Reference rotationSlider}"
        Text="{Binding Value, StringFormat='Rotation = {0:F0}'}"
        Grid.Row="2" Grid.Column="1"
        VerticalTextAlignment="Center" />

    <!-- Slider and identifying Label for RotationX -->
    <Slider x:Name="rotationXSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="3" Grid.Column="0"
        Maximum="360"
        Value="{Binding RotationX, Mode=OneWayToSource}" />

    <Label BindingContext="{x:Reference rotationXSlider}"
        Text="{Binding Value, StringFormat='RotationX = {0:F0}'}"
        Grid.Row="3" Grid.Column="1"
        VerticalTextAlignment="Center" />

    <!-- Slider and identifying Label for RotationY -->
    <Slider x:Name="rotationYSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="4" Grid.Column="0"
        Maximum="360"
        Value="{Binding RotationY, Mode=OneWayToSource}" />

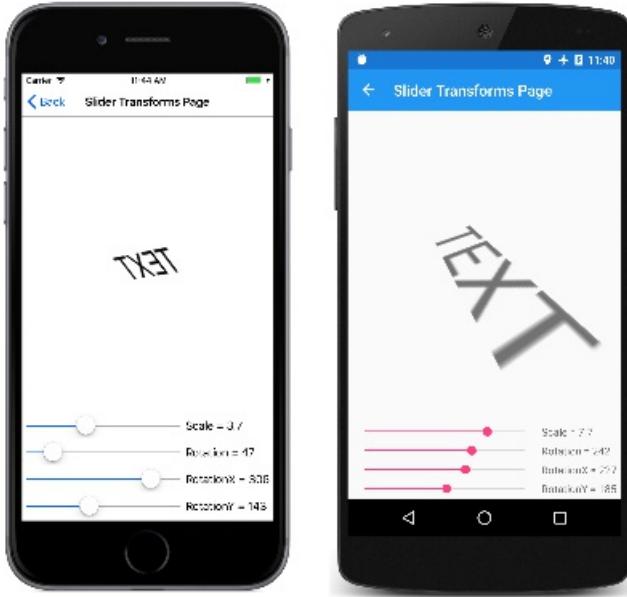
    <Label BindingContext="{x:Reference rotationYSlider}"
        Text="{Binding Value, StringFormat='RotationY = {0:F0}'}"
        Grid.Row="4" Grid.Column="1"
        VerticalTextAlignment="Center" />
</Grid>

```

```
</ContentPage>
```

The bindings on three of the `Slider` views are `OneWayToSource`, meaning that the `Slider` value causes a change in the property of its `BindingContext`, which is the `Label` named `label`. These three `Slider` views cause changes to the `Rotate`, `RotateX`, and `RotateY` properties of the `Label`.

However, the binding for the `Scale` property is `TwoWay`. This is because the `Scale` property has a default value of 1, and using a `TwoWay` binding causes the `Slider` initial value to be set at 1 rather than 0. If that binding were `OneWayToSource`, the `Scale` property would initially be set to 0 from the `Slider` default value. The `Label` would not be visible, and that might cause some confusion to the user.



#### NOTE

The `VisualElement` class also has `ScaleX` and `ScaleY` properties, which scale the `VisualElement` on the x-axis and y-axis respectively.

## Bindings and Collections

Nothing illustrates the power of XAML and data bindings better than a templated `ListView`.

`ListView` defines an `ItemsSource` property of type `IEnumerable`, and it displays the items in that collection. These items can be objects of any type. By default, `ListView` uses the `ToString` method of each item to display that item. Sometimes this is just what you want, but in many cases, `ToString` returns only the fully-qualified class name of the object.

However, the items in the `ListView` collection can be displayed any way you want through the use of a *template*, which involves a class that derives from `Cell`. The template is cloned for every item in the `ListView`, and data bindings that have been set on the template are transferred to the individual clones.

Very often, you'll want to create a custom cell for these items using the `ViewCell` class. This process is somewhat messy in code, but in XAML it becomes very straightforward.

Included in the `XamlSamples` project is a class called `NamedColor`. Each `NamedColor` object has `Name` and `FriendlyName` properties of type `string`, and a `Color` property of type `Color`. In addition, `NamedColor` has 141 static read-only fields of type `Color` corresponding to the colors defined in the `Xamarin.Forms` `Color` class. A static constructor creates an `IEnumerable<NamedColor>` collection that contains `NamedColor` objects corresponding to these static fields, and assigns it to its public static `All` property.

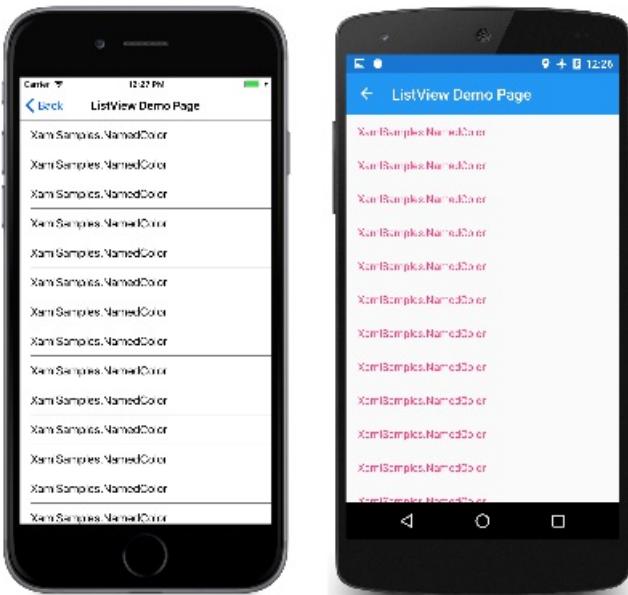
Setting the static `NamedColor.All` property to the `ItemsSource` of a `ListView` is easy using the `x:Static` markup extension:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
    x:Class="XamlSamples.ListViewDemoPage"
    Title="ListView Demo Page">

    <ListView ItemsSource="{x:Static local:NamedColor.All}" />

</ContentPage>
```

The resultant display establishes that the items are truly of type `XamlSamples.NamedColor`:

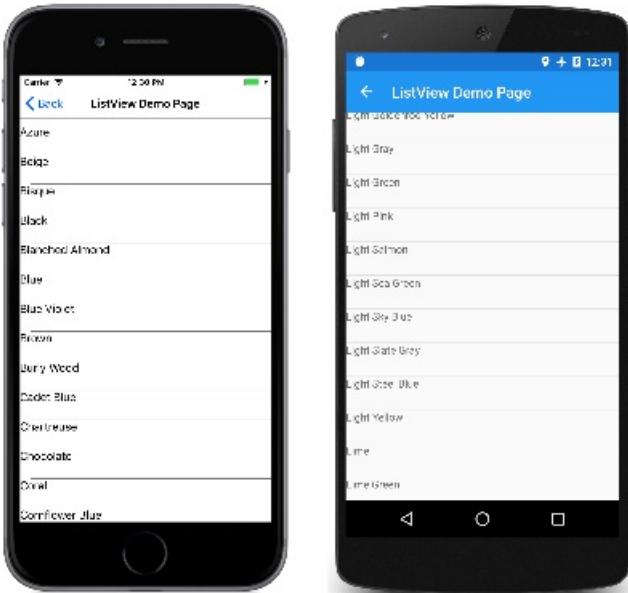


It's not much information, but the `ListView` is scrollable and selectable.

To define a template for the items, you'll want to break out the `ItemTemplate` property as a property element, and set it to a `DataTemplate`, which then references a `ViewCell`. To the `View` property of the `ViewCell` you can define a layout of one or more views to display each item. Here's a simple example:

```
<ListView ItemsSource="{x:Static local:NamedColor.All}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.View>
                    <Label Text="{Binding FriendlyName}" />
                </ViewCell.View>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

The `Label` element is set to the `View` property of the `ViewCell`. (The `ViewCell.View` tags are not needed because the `View` property is the content property of `ViewCell`.) This markup displays the `FriendlyName` property of each `NamedColor` object:



Much better. Now all that's needed is to spruce up the item template with more information and the actual color. To support this template, some values and objects have been defined in the page's resource dictionary:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.ListViewDemoPage"
    Title="ListView Demo Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <OnPlatform x:Key="boxSize"
                x:TypeArguments="x:Double">
                <On Platform="iOS, Android, UWP" Value="50" />
            </OnPlatform>

            <OnPlatform x:Key="rowHeight"
                x:TypeArguments="x:Int32">
                <On Platform="iOS, Android, UWP" Value="60" />
            </OnPlatform>

            <local:DoubleToIntConverter x:Key="intConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView ItemsSource="{x:Static local:NamedColor.All}"
        RowHeight="{StaticResource rowHeight}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <StackLayout Padding="5, 5, 0, 5"
                        Orientation="Horizontal"
                        Spacing="15">

                        <BoxView WidthRequest="{StaticResource boxSize}"
                            HeightRequest="{StaticResource boxSize}"
                            Color="{Binding Color}" />

                        <StackLayout Padding="5, 0, 0, 0"
                            VerticalOptions="Center">

                            <Label Text="{Binding FriendlyName}"
                                FontAttributes="Bold"
                                FontSize="Medium" />
                    </StackLayout>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

```

        <StackLayout Orientation="Horizontal"
                    Spacing="0">
            <Label Text="{Binding Color.R,
                        Converter={StaticResource intConverter},
                        ConverterParameter=255,
                        StringFormat='R={0:X2}'}" />

            <Label Text="{Binding Color.G,
                        Converter={StaticResource intConverter},
                        ConverterParameter=255,
                        StringFormat=', G={0:X2}'}" />

            <Label Text="{Binding Color.B,
                        Converter={StaticResource intConverter},
                        ConverterParameter=255,
                        StringFormat=', B={0:X2}'}" />
        </StackLayout>
    </StackLayout>
</StackLayout>
</ViewCell>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</ContentPage>

```

Notice the use of `OnPlatform` to define the size of a `BoxView` and the height of the `ListView` rows. Although the values for all the platforms are the same, the markup could easily be adapted for other values to fine-tune the display.

## Binding Value Converters

The previous **ListView Demo** XAML file displays the individual `R`, `G`, and `B` properties of the Xamarin.Forms `Color` structure. These properties are of type `double` and range from 0 to 1. If you want to display the hexadecimal values, you can't simply use `StringFormat` with an "X2" formatting specification. That only works for integers and besides, the `double` values need to be multiplied by 255.

This little problem was solved with a *value converter*, also called a *binding converter*. This is a class that implements the `IValueConverter` interface, which means it has two methods named `Convert` and `ConvertBack`. The `Convert` method is called when a value is transferred from source to target; the `ConvertBack` method is called for transfers from target to source in `OneWayToSource` or `TwoWay` bindings:

```

using System;
using System.Globalization;
using Xamarin.Forms;

namespace XamlSamples
{
    class DoubleToIntConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                             object parameter, CultureInfo culture)
        {
            double multiplier;

            if (!Double.TryParse(parameter as string, out multiplier))
                multiplier = 1;

            return (int)Math.Round(multiplier * (double)value);
        }

        public object ConvertBack(object value, Type targetType,
                               object parameter, CultureInfo culture)
        {
            double divider;

            if (!Double.TryParse(parameter as string, out divider))
                divider = 1;

            return ((double)(int)value) / divider;
        }
    }
}

```

The `ConvertBack` method does not play a role in this program because the bindings are only one way from source to target.

A binding references a binding converter with the `Converter` property. A binding converter can also accept a parameter specified with the `ConverterParameter` property. For some versatility, this is how the multiplier is specified. The binding converter checks the converter parameter for a valid `double` value.

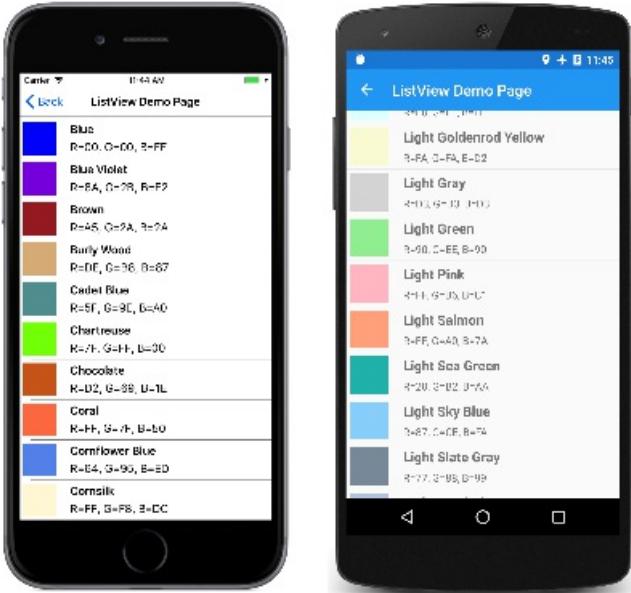
The converter is instantiated in the resource dictionary so it can be shared among multiple bindings:

```
<local:DoubleToIntConverter x:Key="intConverter" />
```

Three data bindings reference this single instance. Notice that the `Binding` markup extension contains an embedded `StaticResource` markup extension:

```
<Label Text="{Binding Color.R,
          Converter={StaticResource intConverter},
          ConverterParameter=255,
          StringFormat='R={0:X2}'}" />
```

Here's the result:



The `ListView` is quite sophisticated in handling changes that might dynamically occur in the underlying data, but only if you take certain steps. If the collection of items assigned to the `ItemsSource` property of the `ListView` changes during runtime—that is, if items can be added to or removed from the collection—use an `ObservableCollection` class for these items. `ObservableCollection` implements the `INotifyCollectionChanged` interface, and `ListView` will install a handler for the `CollectionChanged` event.

If properties of the items themselves change during runtime, then the items in the collection should implement the `INotifyPropertyChanged` interface and signal changes to property values using the `PropertyChanged` event. This is demonstrated in the next part of this series, [Part 5. From Data Binding to MVVM](#).

## Summary

Data bindings provide a powerful mechanism for linking properties between two objects within a page, or between visual objects and underlying data. But when the application begins working with data sources, a popular application architectural pattern begins to emerge as a useful paradigm. This is covered in [Part 5. From Data Bindings to MVVM](#).

## Related Links

- [XamlSamples](#)
- [Part 1. Getting Started with XAML \(sample\)](#)
- [Part 2. Essential XAML Syntax \(sample\)](#)
- [Part 3. XAML Markup Extensions \(sample\)](#)
- [Part 5. From Data Binding to MVVM \(sample\)](#)

# Part 5. From Data Bindings to MVVM

11/12/2018 • 13 minutes to read • [Edit Online](#)

The Model-View-ViewModel (MVVM) architectural pattern was invented with XAML in mind. The pattern enforces a separation between three software layers — the XAML user interface, called the View; the underlying data, called the Model; and an intermediary between the View and the Model, called the ViewModel. The View and the ViewModel are often connected through data bindings defined in the XAML file. The BindingContext for the View is usually an instance of the ViewModel.

## A Simple ViewModel

As an introduction to ViewModels, let's first look at a program without one. Earlier you saw how to define a new XML namespace declaration to allow a XAML file to reference classes in other assemblies. Here's a program that defines an XML namespace declaration for the `System` namespace:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

The program can use `x:Static` to obtain the current date and time from the static `DateTime.Now` property and set that `DateTime` value to the `BindingContext` on a `StackLayout`:

```
<StackLayout BindingContext="{x:Static sys:DateTime.Now}" ...>
```

`BindingContext` is a very special property: When you set the `BindingContext` on an element, it is inherited by all the children of that element. This means that all the children of the `StackLayout` have this same `BindingContext`, and they can contain simple bindings to properties of that object.

In the **One-Shot DateTime** program, two of the children contain bindings to properties of that `DateTime` value, but two other children contain bindings that seem to be missing a binding path. This means that the `DateTime` value itself is used for the `StringFormat`:

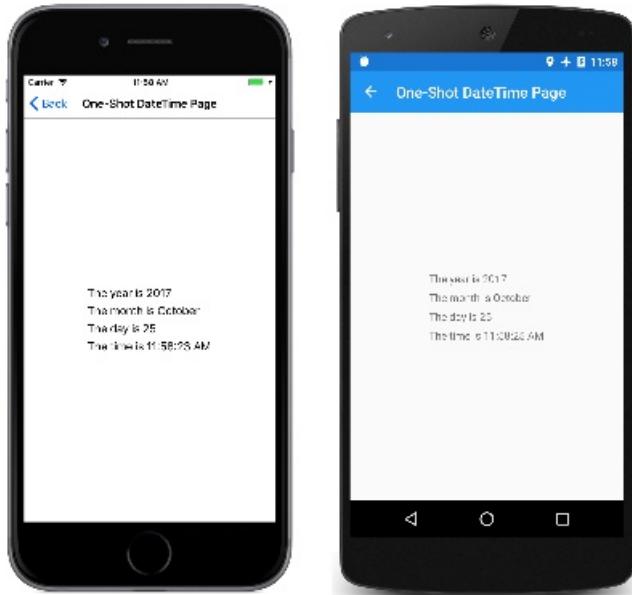
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="XamlSamples.OneShotDateTimePage"
    Title="One-Shot DateTime Page">

    <StackLayout BindingContext="{x:Static sys:DateTime.Now}"
        HorizontalOptions="Center"
        VerticalOptions="Center">

        <Label Text="{Binding Year, StringFormat='The year is {0}'}" />
        <Label Text="{Binding StringFormat='The month is {0:MMMM}'}" />
        <Label Text="{Binding Day, StringFormat='The day is {0}'}" />
        <Label Text="{Binding StringFormat='The time is {0:T}'}" />

    </StackLayout>
</ContentPage>
```

Of course, the big problem is that the date and time are set once when the page is first built, and never change:



A XAML file can display a clock that always shows the current time, but it needs some code to help out. When thinking in terms of MVVM, the Model and ViewModel are classes written entirely in code. The View is often a XAML file that references properties defined in the ViewModel through data bindings.

A proper Model is ignorant of the ViewModel, and a proper ViewModel is ignorant of the View. However, very often a programmer tailors the data types exposed by the ViewModel to the data types associated with particular user interfaces. For example, if a Model accesses a database that contains 8-bit character ASCII strings, the ViewModel would need to convert between those strings to Unicode strings to accommodate the exclusive use of Unicode in the user interface.

In simple examples of MVVM (such as those shown here), often there is no Model at all, and the pattern involves just a View and ViewModel linked with data bindings.

Here's a ViewModel for a clock with just a single property named `DateTime`, but which updates that `DateTime` property every second:

```

using System;
using System.ComponentModel;
using Xamarin.Forms;

namespace XamlSamples
{
    class ClockViewModel : INotifyPropertyChanged
    {
        DateTime dateTime;

        public event PropertyChangedEventHandler PropertyChanged;

        public ClockViewModel()
        {
            this.DateTime = DateTime.Now;

            Device.StartTimer(TimeSpan.FromSeconds(1), () =>
            {
                this.DateTime = DateTime.Now;
                return true;
            });
        }

        public DateTime DateTime
        {
            set
            {
                if (dateTime != value)
                {
                    dateTime = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("DateTime"));
                    }
                }
            }
            get
            {
                return dateTime;
            }
        }
    }
}

```

ViewModels generally implement the `INotifyPropertyChanged` interface, which means that the class fires a `PropertyChanged` event whenever one of its properties changes. The data binding mechanism in Xamarin.Forms attaches a handler to this `PropertyChanged` event so it can be notified when a property changes and keep the target updated with the new value.

A clock based on this ViewModel can be as simple as this:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
    x:Class="XamlSamples.ClockPage"
    Title="Clock Page">

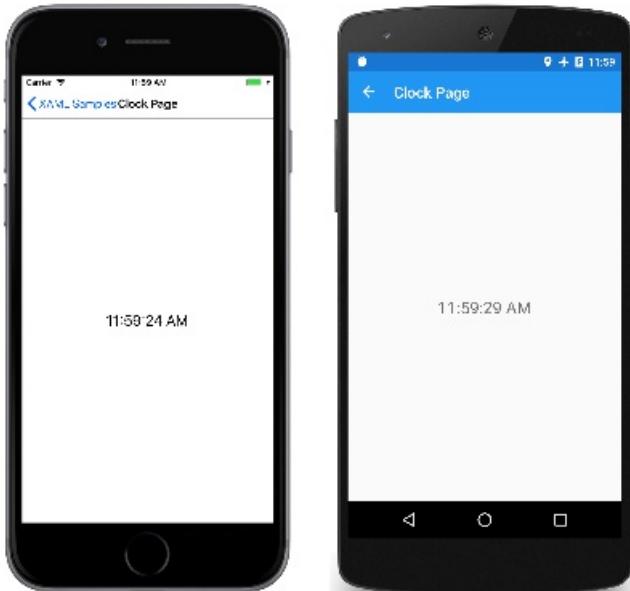
    <Label Text="{Binding DateTime, StringFormat='{0:T}'}"
        FontSize="Large"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <Label.BindingContext>
            <local:ClockViewModel />
        </Label.BindingContext>
    </Label>
</ContentPage>

```

Notice how the `ClockViewModel` is set to the `BindingContext` of the `Label` using property element tags.

Alternatively, you can instantiate the `ClockViewModel` in a `Resources` collection and set it to the `BindingContext` via a `StaticResource` markup extension. Or, the code-behind file can instantiate the ViewModel.

The `Binding` markup extension on the `Text` property of the `Label` formats the `DateTime` property. Here's the display:



It's also possible to access individual properties of the `DateTime` property of the ViewModel by separating the properties with periods:

```
<Label Text="{Binding DateTime.Second, StringFormat='{0}'}" ... >
```

## Interactive MVVM

MVVM is quite often used with two-way data bindings for an interactive view based on an underlying data model.

Here's a class named `HslViewModel` that converts a `Color` value into `Hue`, `Saturation`, and `Luminosity` values, and vice versa:

```

using System;
using System.ComponentModel;
using Xamarin.Forms;

```

```
namespace XamlSamples
{
    public class HslViewModel : INotifyPropertyChanged
    {
        double hue, saturation, luminosity;
        Color color;

        public event PropertyChangedEventHandler PropertyChanged;

        public double Hue
        {
            set
            {
                if (hue != value)
                {
                    hue = value;
                    OnPropertyChanged("Hue");
                    SetNewColor();
                }
            }
            get
            {
                return hue;
            }
        }

        public double Saturation
        {
            set
            {
                if (saturation != value)
                {
                    saturation = value;
                    OnPropertyChanged("Saturation");
                    SetNewColor();
                }
            }
            get
            {
                return saturation;
            }
        }

        public double Luminosity
        {
            set
            {
                if (luminosity != value)
                {
                    luminosity = value;
                    OnPropertyChanged("Luminosity");
                    SetNewColor();
                }
            }
            get
            {
                return luminosity;
            }
        }

        public Color Color
        {
            set
            {
                if (color != value)
                {
                    color = value;
                    OnPropertyChanged("Color");
                }
            }
        }
    }
}
```

```

        Hue = value.Hue;
        Saturation = value.Saturation;
        Luminosity = value.Luminosity;
    }
}
get
{
    return color;
}
}

void SetNewColor()
{
    Color = Color.FromHsla(Hue, Saturation, Luminosity);
}

protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

Changes to the `Hue`, `Saturation`, and `Luminosity` properties cause the `color` property to change, and changes to `color` causes the other three properties to change. This might seem like an infinite loop, except that the class doesn't invoke the `PropertyChanged` event unless the property has actually changed. This puts an end to the otherwise uncontrollable feedback loop.

The following XAML file contains a `BoxView` whose `color` property is bound to the `Color` property of the ViewModel, and three `Slider` and three `Label` views bound to the `Hue`, `Saturation`, and `Luminosity` properties:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
    x:Class="XamlSamples.HslColorScrollPage"
    Title="HSL Color Scroll Page">

<ContentPage.BindingContext>
    <local:HslViewModel Color="Aqua" />
</ContentPage.BindingContext>

<StackLayout Padding="10, 0">
    <BoxView Color="{Binding Color}"
        VerticalOptions="FillAndExpand" />

    <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}"
        HorizontalOptions="Center" />

    <Slider Value="{Binding Hue, Mode=TwoWay}" />

    <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}"
        HorizontalOptions="Center" />

    <Slider Value="{Binding Saturation, Mode=TwoWay}" />

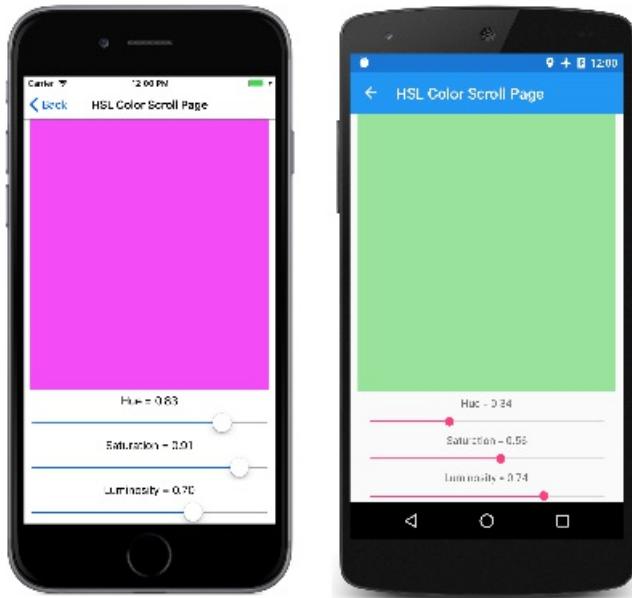
    <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}"
        HorizontalOptions="Center" />

    <Slider Value="{Binding Luminosity, Mode=TwoWay}" />
</StackLayout>
</ContentPage>

```

The binding on each `Label` is the default `oneway`. It only needs to display the value. But the binding on each

`Slider` is `TwoWay`. This allows the `Slider` to be initialized from the ViewModel. Notice that the `Color` property is set to `Aqua` when the ViewModel is instantiated. But a change in the `Slider` also needs to set a new value for the property in the ViewModel, which then calculates a new color.



## Commanding with ViewModels

In many cases, the MVVM pattern is restricted to the manipulation of data items: User-interface objects in the View parallel data objects in the ViewModel.

However, sometimes the View needs to contain buttons that trigger various actions in the ViewModel. But the ViewModel must not contain `Clicked` handlers for the buttons because that would tie the ViewModel to a particular user-interface paradigm.

To allow ViewModels to be more independent of particular user interface objects but still allow methods to be called within the ViewModel, a *command* interface exists. This command interface is supported by the following elements in Xamarin.Forms:

- `Button`
- `MenuItem`
- `ToolbarItem`
- `SearchBar`
- `TextCell` (and hence also `ImageCell`)
- `ListView`
- `TapGestureRecognizer`

With the exception of the `SearchBar` and `ListView` element, these elements define two properties:

- `Command` of type `System.Windows.Input.ICommand`
- `CommandParameter` of type `Object`

The `SearchBar` defines `SearchCommand` and `SearchCommandParameter` properties, while the `ListView` defines a `RefreshCommand` property of type `ICommand`.

The `ICommand` interface defines two methods and one event:

- `void Execute(object arg)`
- `bool CanExecute(object arg)`

- event EventHandler CanExecuteChanged

The ViewModel can define properties of type `ICommand`. You can then bind these properties to the `Command` property of each `Button` or other element, or perhaps a custom view that implements this interface. You can optionally set the `CommandParameter` property to identify individual `Button` objects (or other elements) that are bound to this ViewModel property. Internally, the `Button` calls the `Execute` method whenever the user taps the `Button`, passing to the `Execute` method its `CommandParameter`.

The `CanExecute` method and `CanExecuteChanged` event are used for cases where a `Button` tap might be currently invalid, in which case the `Button` should disable itself. The `Button` calls `CanExecute` when the `Command` property is first set and whenever the `CanExecuteChanged` event is fired. If `CanExecute` returns `false`, the `Button` disables itself and doesn't generate `Execute` calls.

For help in adding commanding to your ViewModels, Xamarin.Forms defines two classes that implement `ICommand`: `Command` and `Command<T>` where `T` is the type of the arguments to `Execute` and `CanExecute`. These two classes define several constructors plus a `ChangeCanExecute` method that the ViewModel can call to force the `Command` object to fire the `CanExecuteChanged` event.

Here is a ViewModel for a simple keypad that is intended for entering telephone numbers. Notice that the `Execute` and `CanExecute` method are defined as lambda functions right in the constructor:

```
using System;
using System.ComponentModel;
using System.Windows.Input;
using Xamarin.Forms;

namespace XamlSamples
{
    class KeypadViewModel : INotifyPropertyChanged
    {
        string inputString = "";
        string displayText = "";
        char[] specialChars = { '*', '#' };

        public event PropertyChangedEventHandler PropertyChanged;

        // Constructor
        public KeypadViewModel()
        {
            AddCharCommand = new Command<string>((key) =>
            {
                // Add the key to the input string.
                InputString += key;
            });

            DeleteCharCommand = new Command(() =>
            {
                // Strip a character from the input string.
                InputString = InputString.Substring(0, InputString.Length - 1);
            },
            () =>
            {
                // Return true if there's something to delete.
                return InputString.Length > 0;
            });
        }

        // Public properties
        public string InputString
        {
            protected set
            {
                if (inputString != value)

```

```

        {
            inputString = value;
            OnPropertyChanged("InputString");
            DisplayText = FormatText(inputString);

            // Perhaps the delete button must be enabled/disabled.
            ((Command)DeleteCharCommand).ChangeCanExecute();
        }
    }

    get { return inputString; }
}

public string DisplayText
{
    protected set
    {
        if (displayText != value)
        {
            displayText = value;
            OnPropertyChanged("DisplayText");
        }
    }
    get { return displayText; }
}

// ICommand implementations
public ICommand AddCharCommand { protected set; get; }

public ICommand DeleteCharCommand { protected set; get; }

string FormatText(string str)
{
    bool hasNonNumbers = str.IndexOfAny(specialChars) != -1;
    string formatted = str;

    if (hasNonNumbers || str.Length < 4 || str.Length > 10)
    {
    }
    else if (str.Length < 8)
    {
        formatted = String.Format("{0}-{1}",
                                   str.Substring(0, 3),
                                   str.Substring(3));
    }
    else
    {
        formatted = String.Format("{0} {1}-{2}",
                                   str.Substring(0, 3),
                                   str.Substring(3, 3),
                                   str.Substring(6));
    }
    return formatted;
}

protected void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

This ViewModel assumes that the `AddCharCommand` property is bound to the `Command` property of several buttons (or anything else that has a command interface), each of which is identified by the `CommandParameter`. These buttons add characters to an `InputString` property, which is then formatted as a phone number for the `DisplayText` property.

There is also a second property of type `ICommand` named `DeleteCharCommand`. This is bound to a back-spacing button, but the button should be disabled if there are no characters to delete.

The following keypad is not as visually sophisticated as it could be. Instead, the markup has been reduced to a minimum to demonstrate more clearly the use of the command interface:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
    x:Class="XamlSamples.KeypadPage"
    Title="Keypad Page">

    <Grid HorizontalOptions="Center"
        VerticalOptions="Center">
        <Grid.BindingContext>
            <local:KeypadViewModel />
        </Grid.BindingContext>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="80" />
            <ColumnDefinition Width="80" />
            <ColumnDefinition Width="80" />
        </Grid.ColumnDefinitions>

        <!-- Internal Grid for top row of items -->
        <Grid Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <Frame Grid.Column="0"
                OutlineColor="Accent">
                <Label Text="{Binding DisplayText}" />
            </Frame>

            <Button Text="⌫"
                Command="{Binding DeleteCharCommand}"
                Grid.Column="1"
                BorderWidth="0" />
        </Grid>

        <Button Text="1"
            Command="{Binding AddCharCommand}"
            CommandParameter="1"
            Grid.Row="1" Grid.Column="0" />

        <Button Text="2"
            Command="{Binding AddCharCommand}"
            CommandParameter="2"
            Grid.Row="1" Grid.Column="1" />

        <Button Text="3"
            Command="{Binding AddCharCommand}"
            CommandParameter="3"
            Grid.Row="1" Grid.Column="2" />

        <Button Text="4"
            Command="{Binding AddCharCommand}">
```

```

        CommandParameter="4"
        Grid.Row="2" Grid.Column="0" />

    <Button Text="5"
        Command="{Binding AddCharCommand}"
        CommandParameter="5"
        Grid.Row="2" Grid.Column="1" />

    <Button Text="6"
        Command="{Binding AddCharCommand}"
        CommandParameter="6"
        Grid.Row="2" Grid.Column="2" />

    <Button Text="7"
        Command="{Binding AddCharCommand}"
        CommandParameter="7"
        Grid.Row="3" Grid.Column="0" />

    <Button Text="8"
        Command="{Binding AddCharCommand}"
        CommandParameter="8"
        Grid.Row="3" Grid.Column="1" />

    <Button Text="9"
        Command="{Binding AddCharCommand}"
        CommandParameter="9"
        Grid.Row="3" Grid.Column="2" />

    <Button Text="*"
        Command="{Binding AddCharCommand}"
        CommandParameter="*"
        Grid.Row="4" Grid.Column="0" />

    <Button Text="0"
        Command="{Binding AddCharCommand}"
        CommandParameter="0"
        Grid.Row="4" Grid.Column="1" />

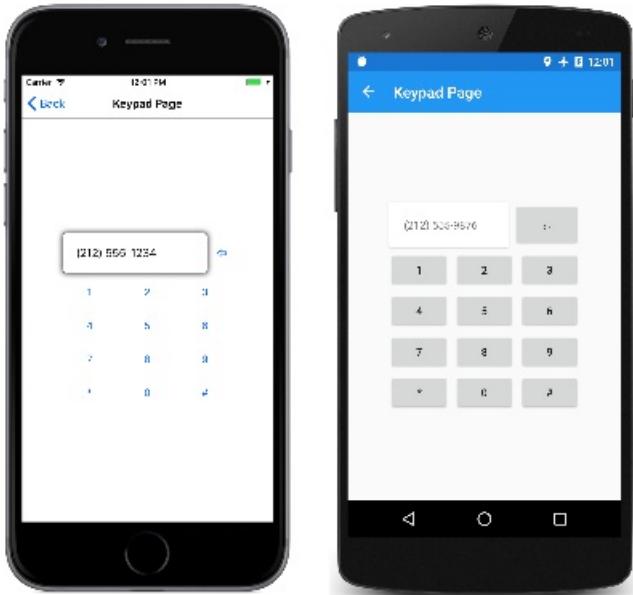
    <Button Text="#"
        Command="{Binding AddCharCommand}"
        CommandParameter="#"
        Grid.Row="4" Grid.Column="2" />

```

</Grid>

</ContentPage>

The `Command` property of the first `Button` that appears in this markup is bound to the `DeleteCharCommand`; the rest are bound to the `AddCharCommand` with a `CommandParameter` that is the same as the character that appears on the `Button` face. Here's the program in action:



## Invoking Asynchronous Methods

Commands can also invoke asynchronous methods. This is achieved by using the `async` and `await` keywords when specifying the `Execute` method:

```
DownloadCommand = new Command (async () => await DownloadAsync ());
```

This indicates that the `DownloadAsync` method is a `Task` and should be awaited:

```
async Task DownloadAsync ()
{
    await Task.Run (() => Download ());
}

void Download ()
{
    ...
}
```

## Implementing a Navigation Menu

The [XamlSamples](#) program that contains all the source code in this series of articles uses a ViewModel for its home page. This ViewModel is a definition of a short class with three properties named `Type`, `Title`, and `Description` that contain the type of each of the sample pages, a title, and a short description. In addition, the ViewModel defines a static property named `All` that is a collection of all the pages in the program:

```
public class PageDataViewModel
{
    public PageDataViewModel(Type type, string title, string description)
    {
        Type = type;
        Title = title;
        Description = description;
    }

    public Type Type { private set; get; }

    public string Title { private set; get; }

    public string Description { private set; get; }
}
```

```

static PageDataViewModel()
{
    All = new List<PageDataViewModel>
    {
        // Part 1. Getting Started with XAML
        new PageDataViewModel(typeof(HelloXamlPage), "Hello, XAML",
            "Display a Label with many properties set"),

        new PageDataViewModel(typeof(XamlPlusCodePage), "XAML + Code",
            "Interact with a Slider and Button"),

        // Part 2. Essential XAML Syntax
        new PageDataViewModel(typeof(GridDemoPage), "Grid Demo",
            "Explore XAML syntax with the Grid"),

        new PageDataViewModel(typeof(AbsoluteDemoPage), "Absolute Demo",
            "Explore XAML syntax with AbsoluteLayout"),

        // Part 3. XAML Markup Extensions
        new PageDataViewModel(typeof(SharedResourcesPage), "Shared Resources",
            "Using resource dictionaries to share resources"),

        new PageDataViewModel(typeof(StaticConstantsPage), "Static Constants",
            "Using the x:Static markup extensions"),

        new PageDataViewModel(typeof(RelativeLayoutPage), "Relative Layout",
            "Explore XAML markup extensions"),

        // Part 4. Data Binding Basics
        new PageDataViewModel(typeof(SliderBindingsPage), "Slider Bindings",
            "Bind properties of two views on the page"),

        new PageDataViewModel(typeof(SliderTransformsPage), "Slider Transforms",
            "Use Sliders with reverse bindings"),

        new PageDataViewModel(typeof(ListViewDemoPage), "ListView Demo",
            "Use a ListView with data bindings"),

        // Part 5. From Data Bindings to MVVM
        new PageDataViewModel(typeof(OneShotDateTimePage), "One-Shot DateTime",
            "Obtain the current DateTime and display it"),

        new PageDataViewModel(typeof(ClockPage), "Clock",
            "Dynamically display the current time"),

        new PageDataViewModel(typeof(HslColorScrollPage), "HSL Color Scroll",
            "Use a view model to select HSL colors"),

        new PageDataViewModel(typeof(KeypadPage), "Keypad",
            "Use a view model for numeric keypad logic")
    };
}

public static IList<PageDataViewModel> All { private set; get; }
}

```

The XAML file for `MainPage` defines a `ListBox` whose `ItemsSource` property is set to that `All` property and which contains a `TextCell` for displaying the `Title` and `Description` properties of each page:

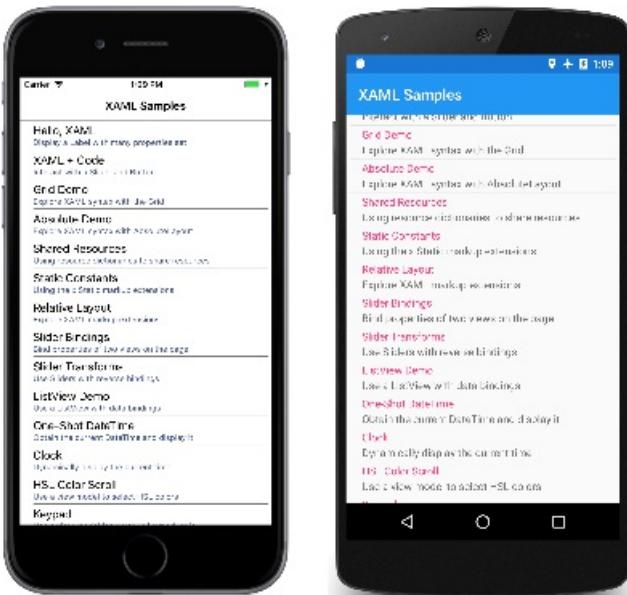
```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.MainPage"
    Padding="5, 0"
    Title="XAML Samples">

    <ListView ItemsSource="{x:Static local:PageDataViewModel.All}"
        ItemSelected="OnListViewItemSelected">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding Title}"
                    Detail="{Binding Description}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>

```

The pages are shown in a scrollable list:



The handler in the code-behind file is triggered when the user selects an item. The handler sets the `SelectedItem` property of the `ListBox` back to `null` and then instantiates the selected page and navigates to it:

```

private async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
{
    (sender as ListView).SelectedItem = null;

    if (args.SelectedItem != null)
    {
        PageDataViewModel pageData = args.SelectedItem as PageDataViewModel;
        Page page = (Page)Activator.CreateInstance(pageData.Type);
        await Navigation.PushAsync(page);
    }
}

```

## Video

**Xamarin Evolve 2016: MVVM Made Simple with Xamarin.Forms and Prism**

## Summary

XAML is a powerful tool for defining user interfaces in Xamarin.Forms applications, particularly when data-binding and MVVM are used. The result is a clean, elegant, and potentially toolable representation of a user interface with all the background support in code.

## Related Links

- [XamlSamples](#)
- [Part 1. Getting Started with XAML](#)
- [Part 2. Essential XAML Syntax](#)
- [Part 3. XAML Markup Extensions](#)
- [Part 4. Data Binding Basics](#)

# XAML Compilation in Xamarin.Forms

11/12/2018 • 2 minutes to read • [Edit Online](#)

XAML can be optionally compiled directly into intermediate language (IL) with the XAML compiler (XAMLC).

XAML compilation offers a number of benefits:

- It performs compile-time checking of XAML, notifying the user of any errors.
- It removes some of the load and instantiation time for XAML elements.
- It helps to reduce the file size of the final assembly by no longer including .xaml files.

XAML compilation is disabled by default to ensure backwards compatibility. It can be enabled at both the assembly and class level by adding the `XamlCompilation` attribute.

The following code example demonstrates enabling XAML compilation at the assembly level:

```
using Xamarin.Forms.Xaml;
...
[assembly: XamlCompilation (XamlCompilationOptions.Compile)]
namespace PhotoApp
{
    ...
}
```

In this example, compile-time checking of all the XAML contained within the assembly will be performed, with XAML errors being reported at compile-time rather than run-time. Therefore, the `assembly` prefix to the `XamlCompilation` attribute specifies that the attribute applies to the entire assembly.

## NOTE

The `XamlCompilation` attribute and the `XamlCompilationOptions` enumeration reside in the `Xamarin.Forms.Xaml` namespace, which must be imported to use them.

The following code example demonstrates enabling XAML compilation at the class level:

```
using Xamarin.Forms.Xaml;
...
[XamlCompilation (XamlCompilationOptions.Compile)]
public class HomePage : ContentPage
{
    ...
}
```

In this example, compile-time checking of the XAML for the `HomePage` class will be performed and errors reported as part of the compilation process.

## NOTE

Compiled bindings can be enabled to improve data binding performance in Xamarin.Forms applications. For more information, see [Compiled Bindings](#).

## Related Links

- [XamlCompilation](#)
- [XamlCompilationOptions](#)

# Xamarin.Forms XAML Toolbox

8/30/2018 • 2 minutes to read • [Edit Online](#)

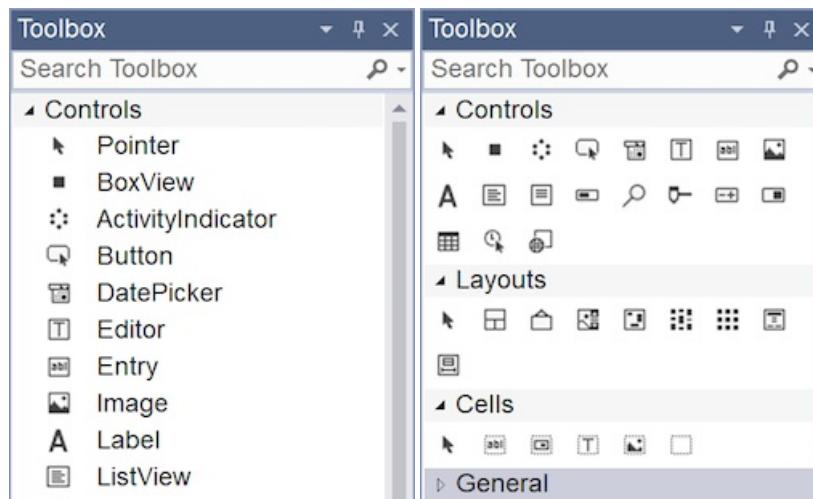
Visual Studio 2017 version 15.8 and Visual Studio for Mac 7.6 now have a Toolbox available while editing Xamarin.Forms XAML files. The toolbox contains all the built-in Xamarin.Forms controls and layouts, which can be dragged into the XAML editor.

- [Visual Studio](#)
- [Visual Studio for Mac](#)

In Visual Studio 2017, open a Xamarin.Forms XAML file for editing. The toolbox can be shown by pressing **Ctrl + W, X** on the keyboard, or choosing the **View > Toolbox** menu item.



The toolbox can be hidden and docked like other panes in Visual Studio 2017, using the icons in the top-right or the context menu. The Xamarin.Forms toolbox has custom view options that can be changed by right-clicking on each section. Toggle the **List View** option to switch between the list and compact views:



When a Xamarin.Forms XAML file is opened for editing, drag any control or layout from the toolbox into the file, then take advantage of Intellisense to customize the user interface.

# XAML Previewer for Xamarin.Forms

11/12/2018 • 2 minutes to read • [Edit Online](#)

*See your Xamarin.Forms layouts rendered as you type!*

## Requirements

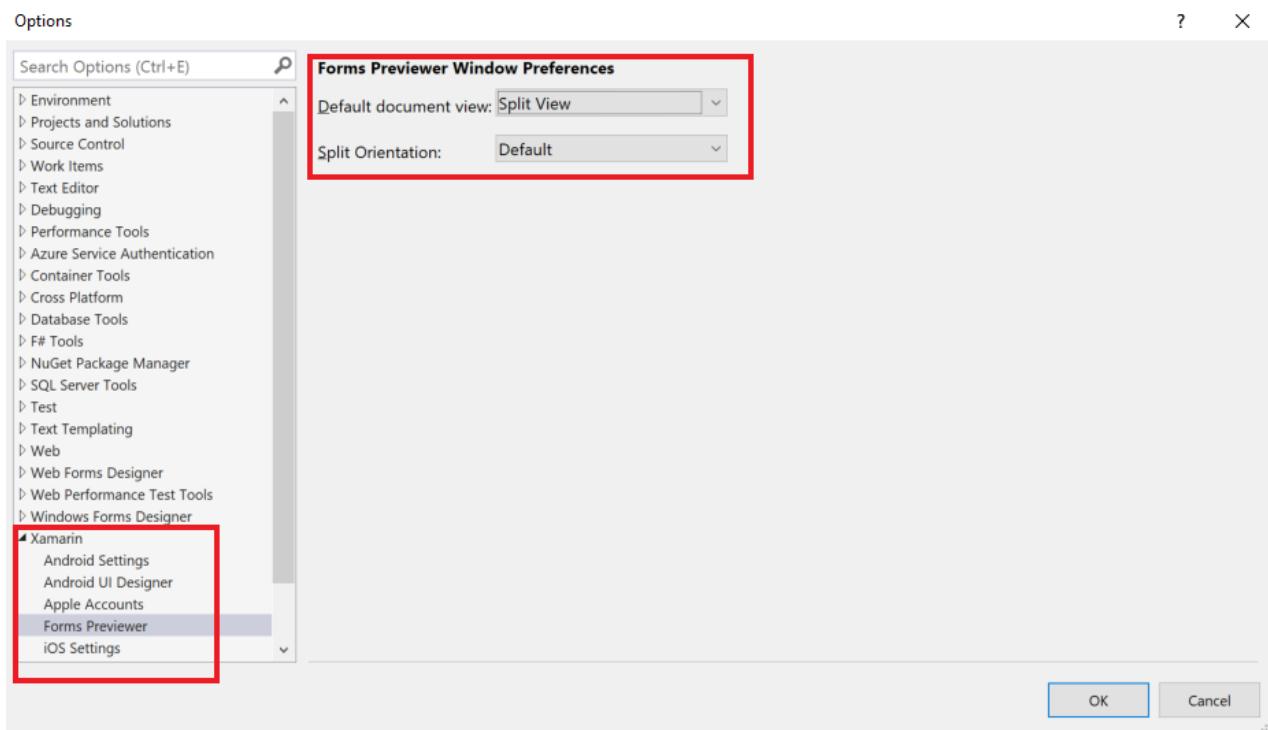
Projects require the latest Xamarin.Forms NuGet package for the XAML Previewer to work. Previewing Android apps requires [JDK 1.8 x64](#).

There is more information in the [release notes](#).

## Getting Started

- [Visual Studio](#)
- [Visual Studio for Mac](#)

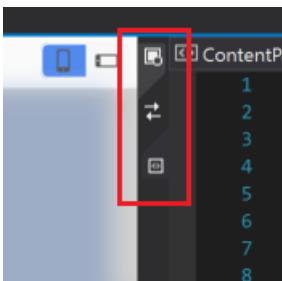
The XAML Previewer is on by default and can be controlled from the **Tools > Options > Xamarin > Forms Previewer** dialog. In this dialog you can select the default document view and the split orientation.



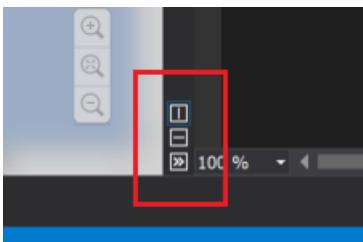
When opening a XAML page the editor will split based on the settings selected in the **Tools > Options > Xamarin > Forms Previewer** dialog. However, these preferences can be changed in the editor window.

## XAML Preview Controls

The top of the editor window has buttons to select which pane is in use, with the top button switching to the design pane and the bottom button switching to the source pane. The middle button swaps the pane order.



The bottom of the editor window has buttons to vertically and horizontally split the panes, and to expand or collapse the current sub-pane.



## XAML Preview Options

The options along the top of the preview pane are:

- **Phone** – render in a phone-size screen
- **Tablet** – render in a tablet-size screen (note there are zoom controls at the bottom-right of the pane)
- **Android** – show the Android version of the screen
- **iOS** – show the iOS version of the screen
- **Portrait** (icon) – uses portrait orientation for the preview
- **Landscape** (icon) – uses landscape orientation for the preview

## Adding Design-Time Data

Some layouts may be hard to visualize without any data bound to the user interface controls. To make the preview more useful, assign some static data to the controls by hardcoding a binding context (either in the code-behind or using XAML).

Refer to James Montemagno's [blog post on adding design-time data](#) to see how to bind to a static ViewModel in XAML.

## Detecting Design Mode

The static `DesignMode.IsEnabled` property can be examined to determine whether the application is running in the previewer. This allows you to specify code that will only execute when the application is running in the previewer:

```
if (DesignMode.IsEnabled)
{
    // Previewer only code
}
```

## Troubleshooting

Check the issues below, and the [Xamarin Forums](#), if you encounter problems.

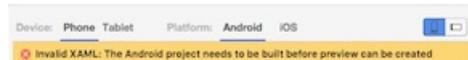
### XAML Preview isn't showing

Check the following:

- Project should be built (compiled) before attempting to preview XAML files.
- The Designer Agent must be set-up the first time you preview a XAML file - a progress indicator will appear in the Previewer, along with progress messages, until this is ready.
- Try closing and re-opening the XAML file.
- Ensure that your `App` class has a parameterless constructor.

#### **Invalid XAML: The Android project needs to be built before preview can be created**

The XAML Previewer requires that the project be built before rendering a page. If the error below appears at the top of the preview pane, re-build the application and try again.



# XAML Namespaces in Xamarin.Forms

11/12/2018 • 3 minutes to read • [Edit Online](#)

XAML uses the `xmlns` XML attribute for namespace declarations. This article introduces the XAML namespace syntax, and demonstrates how to declare a XAML namespace to access a type.

## Overview

There are two XAML namespace declarations that are always within the root element of a XAML file. The first defines the default namespace, as shown in the following XAML code example:

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

The default namespace specifies that elements defined within the XAML file with no prefix refer to Xamarin.Forms classes, such as [ContentPage](#).

The second namespace declaration uses the `x` prefix, as shown in the following XAML code example:

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

XAML uses prefixes to declare non-default namespaces, with the prefix being used when referencing types within the namespace. The `x` namespace declaration specifies that elements defined within the XAML with a prefix of `x` are used for elements and attributes that are intrinsic to XAML (specifically the 2009 XAML specification).

The following table outlines the `x` namespace attributes supported by Xamarin.Forms:

CONSTRUCT	DESCRIPTION
<code>x:Arguments</code>	Specifies constructor arguments for a non-default constructor, or for a factory method object declaration.
<code>x:Class</code>	Specifies the namespace and class name for a class defined in XAML. The class name must match the class name of the code-behind file. Note that this construct can only appear in the root element of a XAML file.
<code>x:DataType</code>	Specifies the type of the object that the XAML element, and its children, will bind to.
<code>x:FactoryMethod</code>	Specifies a factory method that can be used to initialize an object.
<code>x:FieldModifier</code>	Specifies the access level for generated fields for named XAML elements.
<code>x:Key</code>	Specifies a unique user-defined key for each resource in a <a href="#">ResourceDictionary</a> . The key's value is used to retrieve the XAML resource, and is typically used as the argument for the <a href="#">StaticResource</a> markup extension.

CONSTRUCT	DESCRIPTION
x:Name	Specifies a runtime object name for the XAML element. Setting x:Name is similar to declaring a variable in code.
x>TypeArguments	Specifies the generic type arguments to the constructor of a generic type.

For more information about the x:DataType attribute, see [Compiled Bindings](#). For more information about the x:FieldModifier attribute, see [Field Modifiers](#). For more information about the x:Arguments , x:FactoryMethod , and x>TypeArguments attributes, see [Passing Arguments in XAML](#).

In XAML, namespace declarations inherit from parent element to child element. Therefore, when defining a namespace in the root element of a XAML file, all elements within that file inherit the namespace declaration.

## Declaring Namespaces for Types

Types can be referenced in XAML by declaring a XAML namespace with a prefix, with the namespace declaration specifying the Common Language Runtime (CLR) namespace name, and optionally an assembly name. This is achieved by defining values for the following keywords within the namespace declaration:

- **clr-namespace:** or **using:** – the CLR namespace declared within the assembly that contains the types to expose as XAML elements. This keyword is required.
- **assembly=** – the assembly that contains the referenced CLR namespace. This value is the name of the assembly, without the file extension. The path to the assembly should be established as a reference in the project file that contains the XAML file that will reference the assembly. This keyword can be omitted if the **clr-namespace** value is within the same assembly as the application code that's referencing the types.

Note that the character separating the `clr-namespace` or `using` token from its value is a colon, whereas the character separating the `assembly` token from its value is an equal sign. The character to use between the two tokens is a semicolon.

The following code example shows a XAML namespace declaration:

```
<ContentPage ... xmlns:local="clr-namespace:HelloWorld" ...>
  ...
</ContentPage>
```

Alternatively, this can be written as:

```
<ContentPage ... xmlns:local="using:HelloWorld" ...>
  ...
</ContentPage>
```

The `local` prefix is a convention used to indicate that the types within the namespace are local to the application. Alternatively, if the types are in a different assembly, the assembly name should also be defined in the namespace declaration, as demonstrated in the following XAML code example:

```
<ContentPage ... xmlns:behaviors="clr-namespace:Behaviors;assembly=BehaviorsLibrary" ...>
  ...
</ContentPage>
```

The namespace prefix is then specified when declaring an instance of a type from an imported namespace, as demonstrated in the following XAML code example:

```
<ListView ...>
  <ListView.Behaviors>
    <behaviors:EventToCommandBehavior EventName="ItemSelected" ... />
  </ListView.Behaviors>
</ListView>
```

## Summary

This article introduced the XAML namespace syntax, and demonstrated how to declare a XAML namespace to access a type. XAML uses the `xmlns` XML attribute for namespace declarations, and types can be referenced in XAML by declaring a XAML namespace with a prefix.

## Related Links

- [Passing Arguments in XAML](#)

# XAML Markup Extensions

11/12/2018 • 2 minutes to read • [Edit Online](#)

XAML markup extensions help extend the power and flexibility of XAML by allowing element attributes to be set from sources other than literal text strings.

For example, normally you set the `Color` property of `BoxView` like this:

```
<BoxView Color="Blue" />
```

Or, you can set it to a hexadecimal RGB color value:

```
<BoxView Color="#FF0080" />
```

In either case, the text string set to the `Color` attribute is converted to a `Color` value by the `ColorTypeConverter` class.

You might prefer instead to set the `Color` attribute from a value stored in a resource dictionary, or from the value of a static property of a class that you've created, or from a property of type `Color` of another element on the page, or constructed from separate hue, saturation, and luminosity values.

All these options are possible using XAML markup extensions. But don't let the phrase "markup extensions" scare you: XAML markup extensions are *not* extensions to XML. Even with XAML markup extensions, XAML is always legal XML.

A markup extension is really just a different way to express an attribute of an element. XAML markup extensions are usually identifiable by an attribute setting that is enclosed in curly braces:

```
<BoxView Color="{StaticResource themeColor}" />
```

Any attribute setting in curly braces is *always* a XAML markup extension. However, as you'll see, XAML markup extensions can also be referenced without the use of curly braces.

This article is divided in two parts:

## Consuming XAML Markup Extensions

Use the XAML markup extensions defined in Xamarin.Forms.

## Creating XAML Markup Extensions

Write your own custom XAML markup extensions.

## Related Links

- [Markup Extensions \(sample\)](#)
- [XAML markup extensions chapter from Xamarin.Forms book](#)
- [Resource Dictionaries](#)
- [Dynamic Styles](#)

- Data Binding

# Consuming XAML Markup Extensions

11/20/2018 • 14 minutes to read • [Edit Online](#)

XAML markup extensions help enhance the power and flexibility of XAML by allowing element attributes to be set from a variety of sources. Several XAML markup extensions are part of the XAML 2009 specification. These appear in XAML files with the customary `x` namespace prefix, and are commonly referred to with this prefix. This article discusses the following markup extensions:

- `x:Static` – reference static properties, fields, or enumeration members.
- `x:Reference` – reference named elements on the page.
- `x>Type` – set an attribute to a `System.Type` object.
- `x:Array` – construct an array of objects of a particular type.
- `x:Null` – set an attribute to a `null` value.
- `OnPlatform` – customize UI appearance on a per-platform basis.
- `OnIdiom` – customize UI appearance based on the idiom of the device the application is running on.

Additional XAML markup extensions have historically been supported by other XAML implementations, and are also supported by Xamarin.Forms. These are described more fully in other articles:

- `StaticResource` – reference objects from a resource dictionary, as described in the article [Resource Dictionaries](#).
- `DynamicResource` – respond to changes in objects in a resource dictionary, as described in the article [Dynamic Styles](#).
- `Binding` – establish a link between properties of two objects, as described in the article [Data Binding](#).
- `TemplateBinding` – performs data binding from a control template, as discussed in the article [Binding from a Control Template](#).

The `RelativeLayout` layout makes use of the custom markup extension `ConstraintExpression`. This markup extension is described in the article [RelativeLayout](#).

## x:Static Markup Extension

The `x:Static` markup extension is supported by the `StaticExtension` class. The class has a single property named `Member` of type `string` that you set to the name of a public constant, static property, static field, or enumeration member.

One common way to use `x:Static` is to first define a class with some constants or static variables, such as this tiny `AppConstants` class in the [MarkupExtensions](#) program:

```
static class AppConstants
{
    public static double NormalFontSize = 18;
}
```

The [x:Static Demo](#) page demonstrates several ways to use the `x:Static` markup extension. The most verbose approach instantiates the `StaticExtension` class between `Label.FontSize` property-element tags:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.StaticDemoPage"
    Title="x:Static Demo">
    <StackLayout Margin="10, 0">
        <Label Text="Label No. 1">
            <Label.FontSize>
                <x:StaticExtension Member="local:AppConstants.NormalFontSize" />
            </Label.FontSize>
        </Label>
        ...
    </StackLayout>
</ContentPage>

```

The XAML parser also allows the `StaticExtension` class to be abbreviated as `x:Static`:

```

<Label Text="Label No. 2">
    <Label.FontSize>
        <x:Static Member="local:AppConstants.NormalFontSize" />
    </Label.FontSize>
</Label>

```

This can be simplified even further, but the change introduces some new syntax: It consists of putting the `StaticExtension` class and the member setting in curly braces. The resulting expression is set directly to the `FontSize` attribute:

```

<Label Text="Label No. 3"
    FontSize="{x:Static Extension Member=local:AppConstants.NormalFontSize}" />

```

Notice that there are *no* quotation marks within the curly braces. The `Member` property of `StaticExtension` is no longer an XML attribute. It is instead part of the expression for the markup extension.

Just as you can abbreviate `x:StaticExtension` to `x:Static` when you use it as an object element, you can also abbreviate it in the expression within curly braces:

```

<Label Text="Label No. 4"
    FontSize="{x:Static Member=local:AppConstants.NormalFontSize}" />

```

The `StaticExtension` class has a `ContentProperty` attribute referencing the property `Member`, which marks this property as the class's default content property. For XAML markup extensions expressed with curly braces, you can eliminate the `Member=` part of the expression:

```

<Label Text="Label No. 5"
    FontSize="{x:Static local:AppConstants.NormalFontSize}" />

```

This is the most common form of the `x:Static` markup extension.

The **Static Demo** page contains two other examples. The root tag of the XAML file contains an XML namespace declaration for the .NET `System` namespace:

```

xmlns:sys="clr-namespace:System;assembly=mscorlib"

```

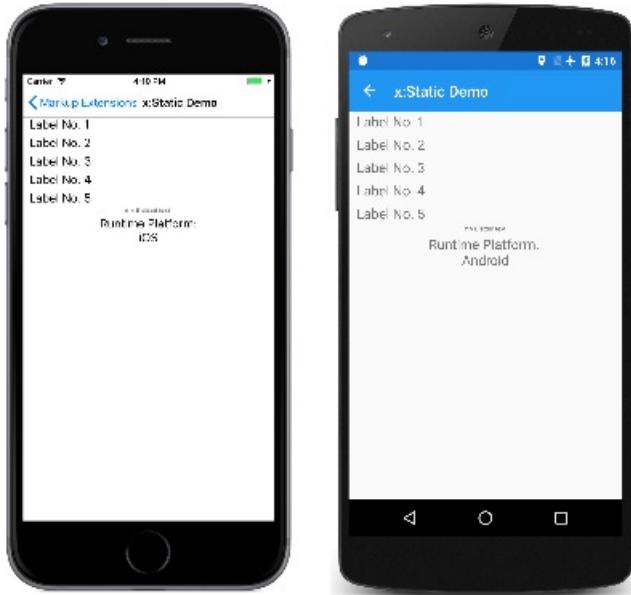
This allows the `Label` font size to be set to the static field `Math.PI`. That results in rather small text, so the `Scale` property is set to `Math.E`:

```
<Label Text="&#x03C0; &#x00D7; E sized text"
    FontSize="{x:Static sys:Math.PI}"
    Scale="{x:Static sys:Math.E}"
    HorizontalOptions="Center" />
```

The final example displays the `Device.RuntimePlatform` value. The `Environment.NewLine` static property is used to insert a new-line character between the two `Span` objects:

```
<Label HorizontalTextAlignment="Center"
    FontSize="{x:Static local:AppConstants.NormalFontSize}">
<Label.FormattedText>
    <FormattedString>
        <Span Text="Runtime Platform: " />
        <Span Text="{x:Static sys:Environment.NewLine}" />
        <Span Text="{x:Static Device.RuntimePlatform}" />
    </FormattedString>
</Label.FormattedText>
</Label>
```

Here's the sample running:



## x:Reference Markup Extension

The `x:Reference` markup extension is supported by the `ReferenceExtension` class. The class has a single property named `Name` of type `string` that you set to the name of an element on the page that has been given a name with `x:Name`. This `Name` property is the content property of `ReferenceExtension`, so `Name=` is not required when `x:Reference` appears in curly braces.

The `x:Reference` markup extension is used exclusively with data bindings, which are described in more detail in the article [Data Binding](#).

The **x:Reference Demo** page shows two uses of `x:Reference` with data bindings, the first where it's used to set the `Source` property of the `Binding` object, and the second where it's used to set the `BindingContext` property for two data bindings:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.ReferenceDemoPage"
    x:Name="page"
    Title="x:Reference Demo">

    <StackLayout Margin="10, 0">

        <Label Text="{Binding Source={x:Reference page},
            StringFormat='The type of this page is {0}'}"
            FontSize="18"
            VerticalOptions="CenterAndExpand"
            HorizontalTextAlignment="Center" />

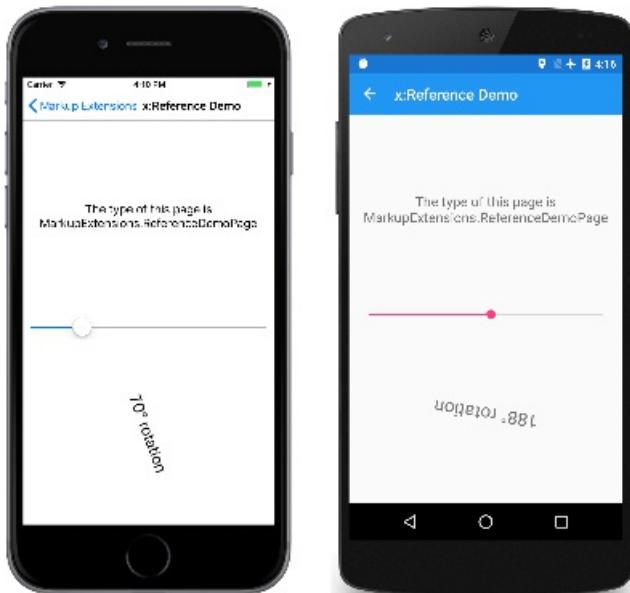
        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="Center" />

        <Label BindingContext="{x:Reference slider}"
            Text="{Binding Value, StringFormat='{0:F0}° rotation'}"
            Rotation="{Binding Value}"
            FontSize="24"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

    </StackLayout>
</ContentPage>

```

Both `x:Reference` expressions use the abbreviated version of the `ReferenceExtension` class name and eliminate the `Name=` part of the expression. In the first example, the `x:Reference` markup extension is embedded in the `Binding` markup extension. Notice that the `Source` and `StringFormat` settings are separated by commas. Here's the program running:



## x:Type Markup Extension

The `x:Type` markup extension is the XAML equivalent of the C# `typeof` keyword. It is supported by the `TypeExtension` class, which defines one property named `TypeName` of type `string` that is set to a class or structure name. The `x:Type` markup extension returns the `System.Type` object of that class or structure. `TypeName` is the content property of `TypeExtension`, so `TypeName=` is not required when `x:Type` appears with curly braces.

Within Xamarin.Forms, there are several properties that have arguments of type `Type`. Examples include the

`TargetType` property of `Style`, and the `x:TypeArguments` attribute used to specify arguments in generic classes. However, the XAML parser performs the `typeof` operation automatically, and the `x:Type` markup extension is not used in these cases.

One place where `x:Type` is required is with the `x:Array` markup extension, which is described in the [next section](#).

The `x:Type` markup extension is also useful when constructing a menu where each menu item corresponds to an object of a particular type. You can associate a `Type` object with each menu item, and then instantiate the object when the menu item is selected.

This is how the navigation menu in `MainPage` in the **Markup Extensions** program works. The `MainPage.xaml` file contains a `TableView` with each `TextCell` corresponding to a particular page in the program:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.MainPage"
    Title="Markup Extensions"
    Padding="10">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection>
                <TextCell Text="x:Static Demo"
                    Detail="Access constants or statics"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:StaticDemoPage}" />

                <TextCell Text="x:Reference Demo"
                    Detail="Reference named elements on the page"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ReferenceDemoPage}" />

                <TextCell Text="x:Type Demo"
                    Detail="Associate a Button with a Type"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:TypeDemoPage}" />

                <TextCell Text="x:Array Demo"
                    Detail="Use an array to fill a ListView"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ArrayDemoPage}" />

                ...
            </TableRoot>
        </TableView>
    </ContentPage>
```

Here's the opening main page in **Markup Extensions**:



Each `CommandParameter` property is set to an `x:Type` markup extension that references one of the other pages. The `Command` property is bound to a property named `NavigateCommand`. This property is defined in the `MainPage` code-behind file:

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(async (Type pageType) =>
        {
            Page page = (Page)Activator.CreateInstance(pageType);
            await Navigation.PushAsync(page);
        });
    }

    BindingContext = this;
}

public ICommand NavigateCommand { private set; get; }
```

The `NavigateCommand` property is a `Command` object that implements an execute command with an argument of type `Type` — the value of `CommandParameter`. The method uses `Activator.CreateInstance` to instantiate the page and then navigates to it. The constructor concludes by setting the `BindingContext` of the page to itself, which enables the `Binding` on `Command` to work. See the [Data Binding](#) article and particularly the [Commanding](#) article for more details about this type of code.

The **x:Type Demo** page uses a similar technique to instantiate `Xamarin.Forms` elements and to add them to a `StackLayout`. The XAML file initially consists of three `Button` elements with their `Command` properties set to a `Binding` and the `CommandParameter` properties set to types of three `Xamarin.Forms` views:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.TypeDemoPage"
    Title="x:Type Demo">

    <StackLayout x:Name="stackLayout"
        Padding="10, 0">

        <Button Text="Create a Slider"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Command="{Binding CreateCommand}"
            CommandParameter="{x:Type Slider}" />

        <Button Text="Create a Stepper"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Command="{Binding CreateCommand}"
            CommandParameter="{x:Type Stepper}" />

        <Button Text="Create a Switch"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Command="{Binding CreateCommand}"
            CommandParameter="{x:Type Switch}" />
    </StackLayout>
</ContentPage>

```

The code-behind file defines and initializes the `CreateCommand` property:

```

public partial class TypeDemoPage : ContentPage
{
    public TypeDemoPage()
    {
        InitializeComponent();

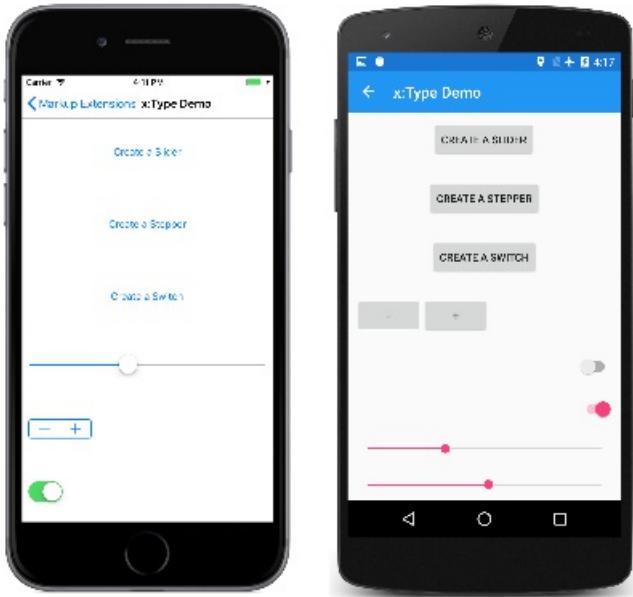
        CreateCommand = new Command<Type>((Type viewType) =>
    {
        View view = (View)Activator.CreateInstance(viewType);
        view.VerticalOptions = LayoutOptions.CenterAndExpand;
        stackLayout.Children.Add(view);
    });
    }

    BindingContext = this;
}

public ICommand CreateCommand { private set; get; }
}

```

The method that is executed when a `Button` is pressed creates a new instance of the argument, sets its `VerticalOptions` property, and adds it to the `StackLayout`. The three `Button` elements then share the page with dynamically created views:



## x:Array Markup Extension

The `x:Array` markup extension allows you to define an array in markup. It is supported by the [ArrayExtension](#) class, which defines two properties:

- `Type` of type `Type`, which indicates the type of the elements in the array.
- `Items` of type `IList`, which is a collection of the items themselves. This is the content property of `ArrayExtension`.

The `x:Array` markup extension itself never appears in curly braces. Instead, `x:Array` start and end tags delimit the list of items. Set the `Type` property to an `x>Type` markup extension.

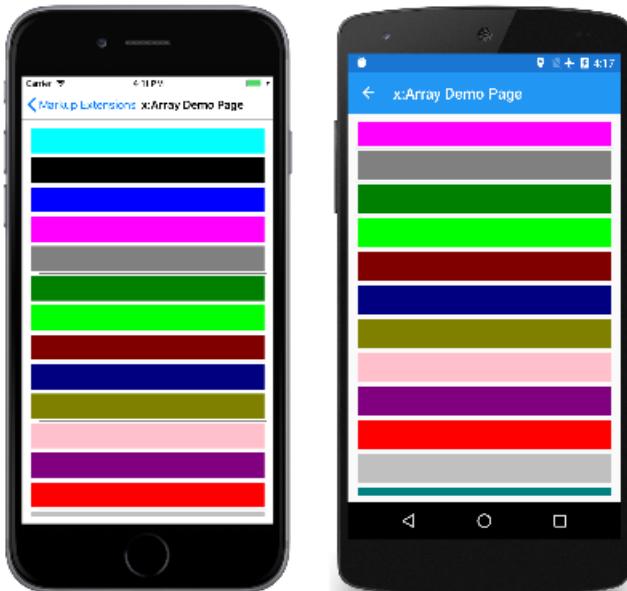
The **x:Array Demo** page shows how to use `x:Array` to add items to a `ListView` by setting the `ItemsSource` property to an array:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.ArrayDemoPage"
    Title="x:Array Demo Page">
    <ListView Margin="10">
        <ListView.ItemsSource>
            <x:Array Type="{x:Type Color}">
                <Color>Aqua</Color>
                <Color>Black</Color>
                <Color>Blue</Color>
                <Color>Fuchsia</Color>
                <Color>Gray</Color>
                <Color>Green</Color>
                <Color>Lime</Color>
                <Color>Maroon</Color>
                <Color>Navy</Color>
                <Color>Olive</Color>
                <Color>Pink</Color>
                <Color>Purple</Color>
                <Color>Red</Color>
                <Color>Silver</Color>
                <Color>Teal</Color>
                <Color>White</Color>
                <Color>Yellow</Color>
            </x:Array>
        </ListView.ItemsSource>
        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <BoxView Color="{Binding}"
                        Margin="3" />
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>

```

The `ViewCell` creates a simple `BoxView` for each color entry:



There are several ways to specify the individual `Color` items in this array. You can use an `x:static` markup extension:

```
<x:Static Member="Color.Blue" />
```

Or, you can use `StaticResource` to retrieve a color from a resource dictionary:

```
<StaticResource Key="myColor" />
```

Towards the end of this article, you'll see a custom XAML markup extension that also creates a new color value:

```
<local:HslColor H="0.5" S="1.0" L="0.5" />
```

When defining arrays of common types like strings or numbers, use the tags listed in the [Passing Constructor Arguments](#) article to delimit the values.

## x:Null Markup Extension

The `x:Null` markup extension is supported by the `NullExtension` class. It has no properties and is simply the XAML equivalent of the C# `null` keyword.

The `x:Null` markup extension is rarely needed and seldom used, but if you do find a need for it, you'll be glad that it exists.

The [x:Null Demo](#) page illustrates one scenario when `x:Null` might be convenient. Suppose that you define an implicit `Style` for `Label` that includes a `Setter` that sets the `FontFamily` property to a platform-dependent family name:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.NullDemoPage"
    Title="x:Null Demo">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="48" />
                <Setter Property="FontFamily">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="x:String">
                            <On Platform="iOS" Value="Times New Roman" />
                            <On Platform="Android" Value="serif" />
                            <On Platform="UWP" Value="Times New Roman" />
                        </OnPlatform>
                    </Setter.Value>
                </Setter>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ContentPage.Content>
        <StackLayout Padding="10, 0">
            <Label Text="Text 1" />
            <Label Text="Text 2" />

            <Label Text="Text 3"
                FontFamily="{x:Null}" />

            <Label Text="Text 4" />
            <Label Text="Text 5" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

Then you discover that for one of the `Label` elements, you want all the property settings in the implicit `Style` except for the `FontFamily`, which you want to be the default value. You could define another `Style` for that purpose but a simpler approach is simply to set the `FontFamily` property of the particular `Label` to `x:Null`, as demonstrated in the center `Label`.

Here's the program running:



Notice that four of the `Label` elements have a serif font, but the center `Label` has the default sans-serif font.

# OnPlatform Markup Extension

The `OnPlatform` markup extension allows you to customize UI appearance on a per-platform basis. It provides the same functionality as the `OnPlatform` and `On` classes, but with a more concise representation.

The `OnPlatform` markup extension is supported by the `OnPlatformExtension` class, which defines the following properties:

- `Default` of type `object`, that you set to a default value to be applied to the properties that represent platforms.
- `Android` of type `object`, that you set to a value to be applied on Android.
- `GTK` of type `object`, that you set to a value to be applied on GTK platforms.
- `iOS` of type `object`, that you set to a value to be applied on iOS.
- `macOS` of type `object`, that you set to a value to be applied on macOS.
- `Tizen` of type `object`, that you set to a value to be applied on the Tizen platform.
- `UWP` of type `object`, that you set to a value to be applied on the Universal Windows Platform.
- `WPF` of type `object`, that you set to a value to be applied on the Windows Presentation Foundation platform.
- `Converter` of type `IValueConverter`, that you set to an `IValueConverter` implementation.
- `ConverterParameter` of type `object`, that you set to a value to pass to the `IValueConverter` implementation.

## NOTE

The XAML parser allows the `OnPlatformExtension` class to be abbreviated as `OnPlatform`.

The `Default` property is the content property of `OnPlatformExtension`. Therefore, for XAML markup expressions expressed with curly braces, you can eliminate the `Default=` part of the expression provided that it's the first argument.

## IMPORTANT

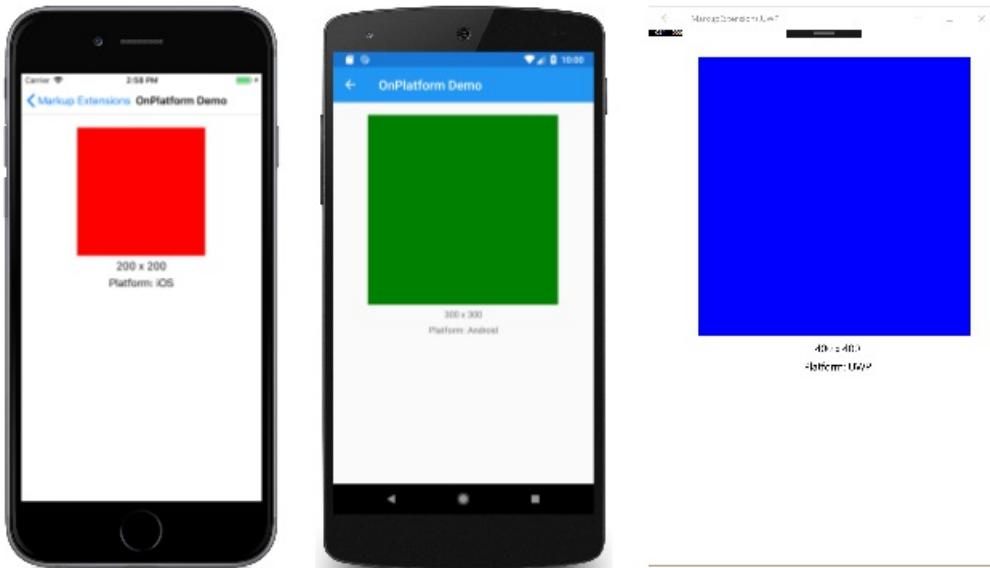
The XAML parser expects that values of the correct type will be provided to properties consuming the `OnPlatform` markup extension. If type conversion is necessary, the `OnPlatform` markup extension will attempt to perform it using the default converters provided by Xamarin.Forms. However, there are some type conversions that can't be performed by the default converters and in these cases the `Converter` property should be set to an `IValueConverter` implementation.

The **OnPlatform Demo** page shows how to use the `OnPlatform` markup extension:

```
<BoxView Color="{OnPlatform Yellow, iOS=Red, Android=Green, UWP=Blue}"  
        WidthRequest="{OnPlatform 250, iOS=200, Android=300, UWP=400}"  
        HeightRequest="{OnPlatform 250, iOS=200, Android=300, UWP=400}"  
        HorizontalOptions="Center" />
```

In this example, all three `OnPlatform` expressions use the abbreviated version of the `OnPlatformExtension` class name. The three `OnPlatform` markup extensions set the `Color`, `WidthRequest`, and `HeightRequest` properties of the `BoxView` to different values on iOS, Android, and UWP. The markup extensions also provide default values for these properties on the platforms that aren't specified, while eliminating the `Default=` part of the expression. Notice that the markup extension properties that are set are separated by commas.

Here's the program running:



## OnIdiom Markup Extension

The `OnIdiom` markup extensions allows you to customize UI appearance based on the idiom of the device the application is running on. It's supported by the `OnIdiomExtension` class, which defines the following properties:

- `Default` of type `object`, that you set to a default value to be applied to the properties that represent device idioms.
- `Phone` of type `object`, that you set to a value to be applied on phones.
- `Tablet` of type `object`, that you set to a value to be applied on tablets.
- `Desktop` of type `object`, that you set to a value to be applied on desktop platforms.
- `TV` of type `object`, that you set to a value to be applied on TV platforms.
- `Watch` of type `object`, that you set to a value to be applied on Watch platforms.
- `Converter` of type `IValueConverter`, that you set to an `IValueConverter` implementation.
- `ConverterParameter` of type `object`, that you set to a value to pass to the `IValueConverter` implementation.

### NOTE

The XAML parser allows the `OnIdiomExtension` class to be abbreviated as `OnIdiom`.

The `Default` property is the content property of `OnIdiomExtension`. Therefore, for XAML markup expressions expressed with curly braces, you can eliminate the `Default=` part of the expression provided that it's the first argument.

### IMPORTANT

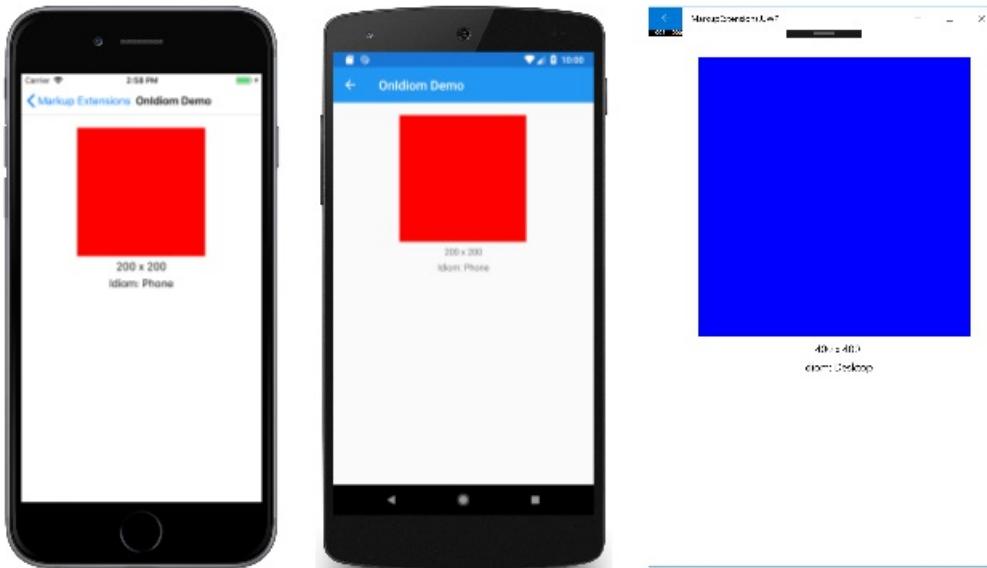
The XAML parser expects that values of the correct type will be provided to properties consuming the `OnIdiom` markup extension. If type conversion is necessary, the `OnIdiom` markup extension will attempt to perform it using the default converters provided by Xamarin.Forms. However, there are some type conversions that can't be performed by the default converters and in these cases the `Converter` property should be set to an `IValueConverter` implementation.

The **OnIdiom Demo** page shows how to use the `OnIdiom` markup extension:

```
<BoxView Color="{OnIdiom Yellow, Phone=Red, Tablet=Green, Desktop=Blue}"  
        WidthRequest="{OnIdiom 100, Phone=200, Tablet=300, Desktop=400}"  
        HeightRequest="{OnIdiom 100, Phone=200, Tablet=300, Desktop=400}"  
        HorizontalOptions="Center" />
```

In this example, all three `OnIdiom` expressions use the abbreviated version of the `OnIdiomExtension` class name. The three `OnIdiom` markup extensions set the `Color`, `WidthRequest`, and `HeightRequest` properties of the `BoxView` to different values on the phone, tablet, and desktop idioms. The markup extensions also provide default values for these properties on the idioms that aren't specified, while eliminating the `Default=` part of the expression. Notice that the markup extension properties that are set are separated by commas.

Here's the program running:



## Define Your Own Markup Extensions

If you've encountered a need for a XAML markup extension that isn't available in Xamarin.Forms, you can [create your own](#).

## Related Links

- [Markup Extensions \(sample\)](#)
- [XAML markup extensions chapter from Xamarin.Forms book](#)
- [Resource Dictionaries](#)
- [Dynamic Styles](#)
- [Data Binding](#)

# Creating XAML Markup Extensions

11/20/2018 • 5 minutes to read • [Edit Online](#)

On the programmatic level, a XAML markup extension is a class that implements the `IMarkupExtension` or `IMarkupExtension<T>` interface. You can explore the source code of the standard markup extensions described below in the [MarkupExtensions directory](#) of the Xamarin.Forms GitHub repository.

It's also possible to define your own custom XAML markup extensions by deriving from `IMarkupExtension` or `IMarkupExtension<T>`. Use the generic form if the markup extension obtains a value of a particular type. This is the case with several of the Xamarin.Forms markup extensions:

- `TypeExtension` derives from `IMarkupExtension<Type>`
- `ArrayExtension` derives from `IMarkupExtension<Array>`
- `DynamicResourceExtension` derives from `IMarkupExtension<DynamicResource>`
- `BindingExtension` derives from `IMarkupExtension<BindingBase>`
- `ConstraintExpression` derives from `IMarkupExtension<Constraint>`

The two `IMarkupExtension` interfaces define only one method each, named `ProvideValue`:

```
public interface IMarkupExtension
{
    object ProvideValue(IServiceProvider serviceProvider);
}

public interface IMarkupExtension<out T> : IMarkupExtension
{
    new T ProvideValue(IServiceProvider serviceProvider);
}
```

Since `IMarkupExtension<T>` derives from `IMarkupExtension` and includes the `new` keyword on `ProvideValue`, it contains both `ProvideValue` methods.

Very often, XAML markup extensions define properties that contribute to the return value. (The obvious exception is `NullExtension`, in which `ProvideValue` simply returns `null`.) The `ProvideValue` method has a single argument of type `IServiceProvider` that will be discussed later in this article.

## A Markup Extension for Specifying Color

The following XAML markup extension allows you to construct a `Color` value using hue, saturation, and luminosity components. It defines four properties for the four components of the color, including an alpha component that is initialized to 1. The class derives from `IMarkupExtension<Color>` to indicate a `Color` return value:

```

public class HslColorExtension : IMarkupExtension<Color>
{
    public double H { set; get; }

    public double S { set; get; }

    public double L { set; get; }

    public double A { set; get; } = 1.0;

    public Color ProvideValue(IServiceProvider serviceProvider)
    {
        return Color.FromHsla(H, S, L, A);
    }

    object IMarkupExtension.ProvideValue(IServiceProvider serviceProvider)
    {
        return (this as IMarkupExtension<Color>).ProvideValue(serviceProvider);
    }
}

```

Because `IMarkupExtension<T>` derives from `IMarkupExtension`, the class must contain two `ProvideValue` methods, one that returns `Color` and another that returns `object`, but the second method can simply call the first method.

The **HSL Color Demo** page shows a variety of ways that `HslColorExtension` can appear in a XAML file to specify the color for a `BoxView`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.HslColorDemoPage"
    Title="HSL Color Demo">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="WidthRequest" Value="80" />
                <Setter Property="HeightRequest" Value="80" />
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <BoxView>
            <BoxView.Color>
                <local:HslColorExtension H="0" S="1" L="0.5" A="1" />
            </BoxView.Color>
        </BoxView>

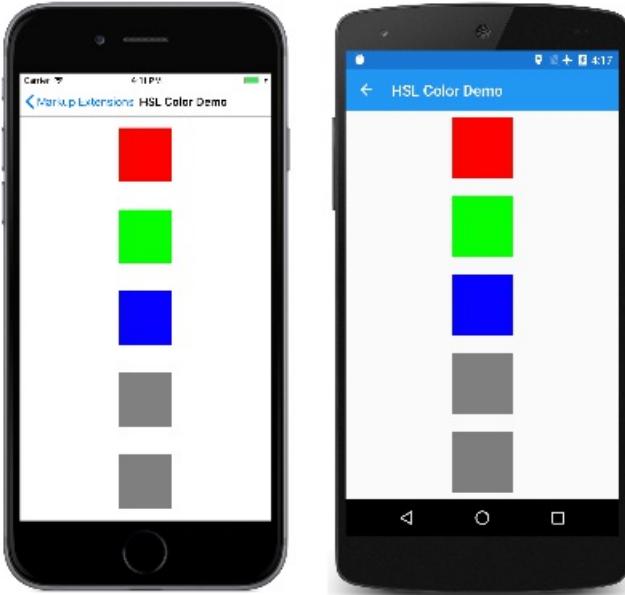
        <BoxView>
            <BoxView.Color>
                <local:HslColor H="0.33" S="1" L="0.5" />
            </BoxView.Color>
        </BoxView>

        <BoxView Color="{local:HslColorExtension H=0.67, S=1, L=0.5}" />
        <BoxView Color="{local:HslColor H=0, S=0, L=0.5}" />

        <BoxView Color="{local:HslColor A=0.5}" />
    </StackLayout>
</ContentPage>

```

Notice that when `HslColorExtension` is an XML tag, the four properties are set as attributes, but when it appears between curly braces, the four properties are separated by commas without quotation marks. The default values for `H`, `S`, and `L` are 0, and the default value of `A` is 1, so those properties can be omitted if you want them set to default values. The last example shows an example where the luminosity is 0, which normally results in black, but the alpha channel is 0.5, so it is half transparent and appears gray against the white background of the page:



## A Markup Extension for Accessing Bitmaps

The argument to `ProvideValue` is an object that implements the `IServiceProvider` interface, which is defined in the .NET `System` namespace. This interface has one member, a method named `GetService` with a `Type` argument.

The `ImageResourceExtension` class shown below shows one possible use of `IServiceProvider` and `GetService` to obtain an `IXmlAttributeInfoProvider` object that can provide line and character information indicating where a particular error was detected. In this case, an exception is raised when the `Source` property has not been set:

```

[ContentProperty("Source")]
class ImageResourceExtension : IMarkupExtension<ImageSource>
{
    public string Source { set; get; }

    public ImageSource ProvideValue(IServiceProvider serviceProvider)
    {
        if (String.IsNullOrEmpty(Source))
        {
            IXmlLineInfoProvider lineInfoProvider = serviceProvider.GetService(typeof(IXmlLineInfoProvider)) as IXmlLineInfoProvider;
            IXmlLineInfo lineInfo = (lineInfoProvider != null) ? lineInfoProvider.XmlLineInfo : new XmlLineInfo();
            throw new XamlParseException("ImageResourceExtension requires Source property to be set",
                lineInfo);
        }

        string assemblyName = GetType().GetTypeInfo().Assembly.GetName().Name;
        return ImageSource.FromResource(assemblyName + "." + Source,
            typeof(ImageResourceExtension).GetTypeInfo().Assembly);
    }

    object IMarkupExtension.ProvideValue(IServiceProvider serviceProvider)
    {
        return (this as IMarkupExtension<ImageSource>).ProvideValue(serviceProvider);
    }
}

```

`ImageResourceExtension` is helpful when a XAML file needs to access an image file stored as an embedded resource in the .NET Standard library project. It uses the `Source` property to call the static `ImageSource.FromResource` method. This method requires a fully-qualified resource name, which consists of the assembly name, the folder name, and the filename separated by periods. The second argument to the `ImageSource.FromResource` method provides the assembly name, and is only required for release builds on UWP. Regardless, `ImageSource.FromResource` must be called from the assembly that contains the bitmap, which means that this XAML resource extension cannot be part of an external library unless the images are also in that library. (See the [Embedded Images](#) article for more information on accessing bitmaps stored as embedded resources.)

Although `ImageResourceExtension` requires the `Source` property to be set, the `Source` property is indicated in an attribute as the content property of the class. This means that the `Source=` part of the expression in curly braces can be omitted. In the [Image Resource Demo](#) page, the `Image` elements fetch two images using the folder name and the filename separated by periods:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.ImageResourceDemoPage"
    Title="Image Resource Demo">

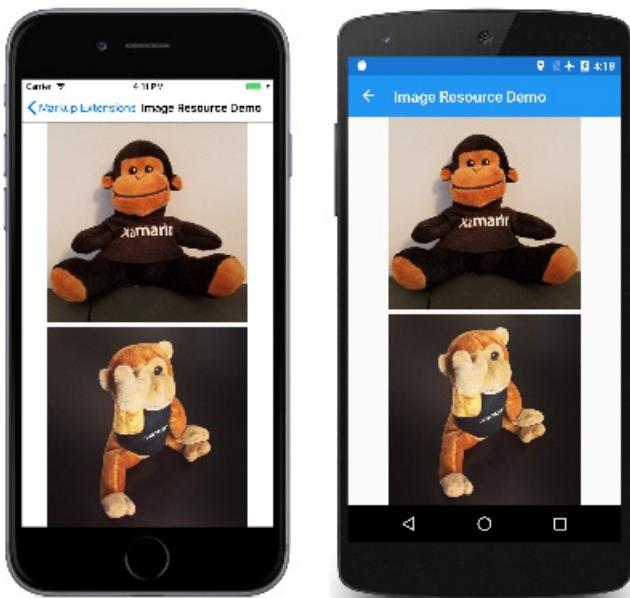
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Image Source="{local:ImageResource Images.SeatedMonkey.jpg}"
            Grid.Row="0" />

        <Image Source="{local:ImageResource Images.FacePalm.jpg}"
            Grid.Row="1" />
    </Grid>
</ContentPage>

```

Here's the program running:



## Service Providers

By using the `IServiceProvider` argument to `ProvideValue`, XAML markup extensions can get access to helpful information about the XAML file in which they're being used. But to use the `IServiceProvider` argument successfully, you need to know what kind of services are available in particular contexts. The best way to get an understanding of this feature is by studying the source code of existing XAML markup extensions in the [MarkupExtensions folder](#) in the Xamarin.Forms repository on GitHub. Be aware that some types of services are internal to Xamarin.Forms.

In some XAML markup extensions, this service might be useful:

```
IProvideValueTarget provideValueTarget = serviceProvider.GetService(typeof(IProvideValueTarget)) as IProvideValueTarget;
```

The `IProvideValueTarget` interface defines two properties, `TargetObject` and `TargetProperty`. When this information is obtained in the `ImageResourceExtension` class, `TargetObject` is the `Image` and `TargetProperty` is a `BindableProperty` object for the `Source` property of `Image`. This is the property on which the XAML markup extension has been set.

The `GetService` call with an argument of `typeof(IProvideValueTarget)` actually returns an object of type `SimpleValueTargetProvider`, which is defined in the `Xamarin.Forms.Xaml.Internals` namespace. If you cast the return value of `GetService` to that type, you can also access a `ParentObjects` property, which is an array that contains the `Image` element, the `Grid` parent, and the `ImageResourceDemoPage` parent of the `Grid`.

## Conclusion

XAML markup extensions play a vital role in XAML by extending the ability to set attributes from a variety of sources. Moreover, if the existing XAML markup extensions don't provide exactly what you need, you can also write your own.

## Related Links

- [Markup Extensions \(sample\)](#)
- [XAML markup extensions chapter from Xamarin.Forms book](#)

# XAML Field Modifiers in Xamarin.Forms

6/20/2018 • 2 minutes to read • [Edit Online](#)

The `x:FieldModifier` namespace attribute specifies the access level for generated fields for named XAML elements.

## Overview

Valid values of the attribute are:

- `Public` – specifies that the generated field for the XAML element is `public`.
- `NotPublic` – specifies that the generated field for the XAML element is `internal` to the assembly.

If the value of the attribute isn't set, the generated field for the element will be `private`.

The following conditions must be met for an `x:FieldModifier` attribute to be processed:

- The top-level XAML element must be a valid `x:Class`.
- The current XAML element has an `x:Name` specified.

The following XAML shows examples of setting the attribute:

```
<Label x:Name="privateLabel" />
<Label x:Name="internalLabel" x:FieldModifier="NotPublic" />
<Label x:Name="publicLabel" x:FieldModifier="Public" />
```

### NOTE

The `x:FieldModifier` attribute cannot be used to specify the access level of a XAML class.

# Passing Arguments in XAML

11/12/2018 • 3 minutes to read • [Edit Online](#)

This article demonstrates using the XAML attributes that can be used to pass arguments to non-default constructors, to call factory methods, and to specify the type of a generic argument.

## Overview

It's often necessary to instantiate objects with constructors that require arguments, or by calling a static creation method. This can be achieved in XAML by using the `x:Arguments` and `x:FactoryMethod` attributes:

- The `x:Arguments` attribute is used to specify constructor arguments for a non-default constructor, or for a factory method object declaration. For more information, see [Passing Constructor Arguments](#).
- The `x:FactoryMethod` attribute is used to specify a factory method that can be used to initialize an object. For more information, see [Calling Factory Methods](#).

In addition, the `x>TypeArguments` attribute can be used to specify the generic type arguments to the constructor of a generic type. For more information, see [Specifying a Generic Type Argument](#).

## Passing Constructor Arguments

Arguments can be passed to a non-default constructor using the `x:Arguments` attribute. Each constructor argument must be delimited within an XML element that represents the type of the argument. Xamarin.Forms supports the following elements for basic types:

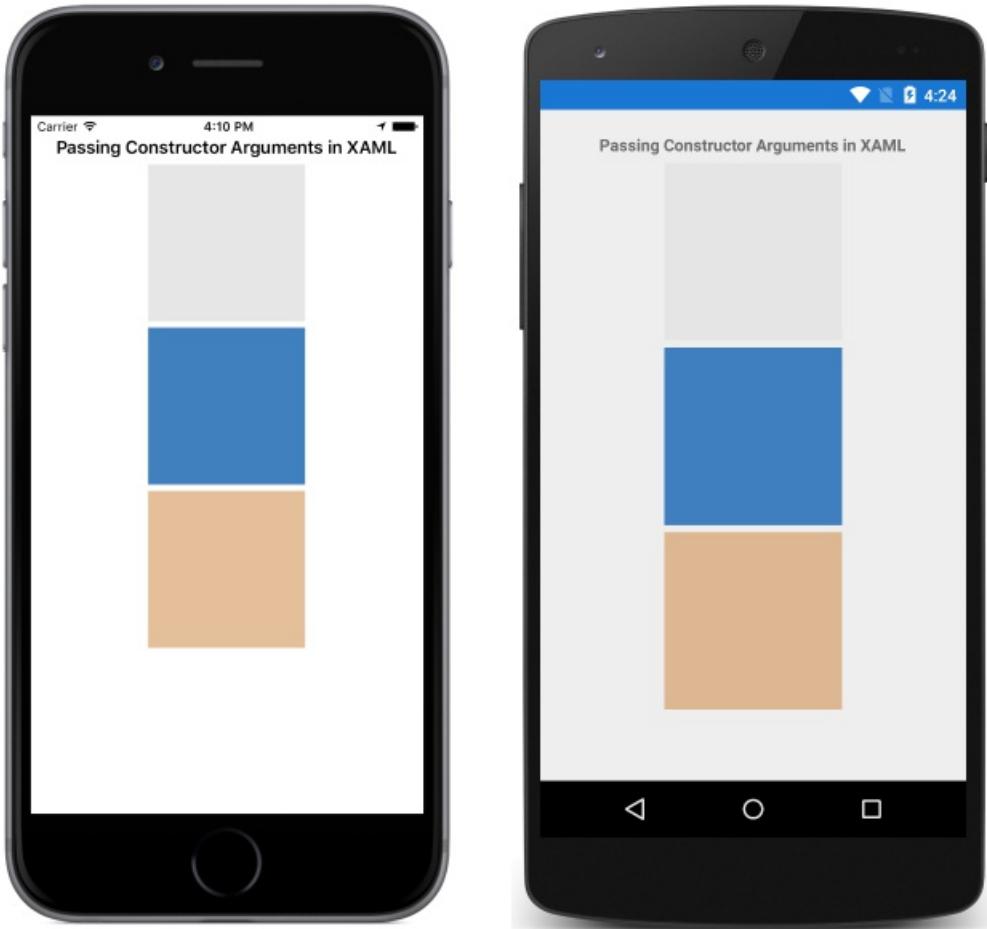
- `x:Object`
- `x:Boolean`
- `x[Byte]`
- `x:Int16`
- `x:Int32`
- `x:Int64`
- `x:Single`
- `x:Double`
- `x:Decimal`
- `x:Char`
- `x:String`
- `x:TimeSpan`
- `x:Array`
- `x:DateTime`

The following code example demonstrates using the `x:Arguments` attribute with three `Color` constructors:

```
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
<BoxView.Color>
<Color>
<x:Arguments>
<x:Double>0.9</x:Double>
</x:Arguments>
</Color>
</BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
<BoxView.Color>
<Color>
<x:Arguments>
<x:Double>0.25</x:Double>
<x:Double>0.5</x:Double>
<x:Double>0.75</x:Double>
</x:Arguments>
</Color>
</BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
<BoxView.Color>
<Color>
<x:Arguments>
<x:Double>0.8</x:Double>
<x:Double>0.5</x:Double>
<x:Double>0.2</x:Double>
<x:Double>0.5</x:Double>
</x:Arguments>
</Color>
</BoxView.Color>
</BoxView>
```

The number of elements within the `x:Arguments` tag, and the types of these elements, must match one of the `Color` constructors. The `color constructor` with a single parameter requires a grayscale value from 0 (black) to 1 (white). The `color constructor` with three parameters requires a red, green, and blue value ranging from 0 to 1. The `color constructor` with four parameters adds an alpha channel as the fourth parameter.

The following screenshots show the result of calling each `color` constructor with the specified argument values:



## Calling Factory Methods

Factory methods can be called in XAML by specifying the method's name using the `x:FactoryMethod` attribute, and its arguments using the `x:Arguments` attribute. A factory method is a `public static` method that returns objects or values of the same type as the class or structure that defines the methods.

The `Color` structure defines a number of factory methods, and the following code example demonstrates calling three of them:

```

<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromRgba">
      <x:Arguments>
        <x:Int32>192</x:Int32>
        <x:Int32>75</x:Int32>
        <x:Int32>150</x:Int32>
        <x:Int32>128</x:Int32>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromHsla">
      <x:Arguments>
        <x:Double>0.23</x:Double>
        <x:Double>0.42</x:Double>
        <x:Double>0.69</x:Double>
        <x:Double>0.7</x:Double>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromHex">
      <x:Arguments>
        <x:String>#FF048B9A</x:String>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>

```

The number of elements within the `x:Arguments` tag, and the types of these elements, must match the arguments of the factory method being called. The `FromRgba` factory method requires four `Int32` parameters, which represent the red, green, blue, and alpha values, ranging from 0 to 255 respectively. The `FromHsla` factory method requires four `Double` parameters, which represent the hue, saturation, luminosity, and alpha values, ranging from 0 to 1 respectively. The `FromHex` factory method requires a `String` that represents the hexadecimal (A)RGB color.

The following screenshots show the result of calling each `Color` factory method with the specified argument values:



## Specifying a Generic Type Argument

Generic type arguments for the constructor of a generic type can be specified using the `x:TypeArguments` attribute, as demonstrated in the following code example:

```
<ContentPage ...>
    <StackLayout>
        <StackLayout.Margin>
            <OnPlatform x:TypeArguments="Thickness">
                <On Platform="iOS" Value="0,20,0,0" />
                <On Platform="Android" Value="5, 10" />
                <On Platform="UWP" Value="10" />
            </OnPlatform>
        </StackLayout.Margin>
    </StackLayout>
</ContentPage>
```

The `OnPlatform` class is a generic class and must be instantiated with an `x:TypeArguments` attribute that matches the target type. In the `On` class, the `Platform` attribute can accept a single `string` value, or multiple comma-delimited `string` values. In this example, the `StackLayout.Margin` property is set to a platform-specific `Thickness`.

## Summary

This article demonstrated using the XAML attributes that can be used to pass arguments to non-default constructors, to call factory methods, and to specify the type of a generic argument.

## Related Links

- [XAML Namespaces](#)
- [Passing Constructor Arguments \(sample\)](#)
- [Calling Factory Methods \(sample\)](#)

# Bindable Properties

11/12/2018 • 8 minutes to read • [Edit Online](#)

In *Xamarin.Forms*, the functionality of common language runtime (CLR) properties is extended by bindable properties. A bindable property is a special type of property, where the property's value is tracked by the *Xamarin.Forms* property system. This article provides an introduction to bindable properties, and demonstrates how to create and consume them.

## Overview

Bindable properties extend CLR property functionality by backing a property with a `BindableProperty` type, instead of backing a property with a field. The purpose of bindable properties is to provide a property system that supports data binding, styles, templates, and values set through parent-child relationships. In addition, bindable properties can provide default values, validation of property values, and callbacks that monitor property changes.

Properties should be implemented as bindable properties to support one or more of the following features:

- Acting as a valid *target* property for data binding.
- Setting the property through a [style](#).
- Providing a default property value that's different from the default for the type of the property.
- Validating the value of the property.
- Monitoring property changes.

Examples of *Xamarin.Forms* bindable properties include `Label.Text`, `Button.BorderRadius`, and `StackLayout.Orientation`. Each bindable property has a corresponding `public static readonly` property of type `BindableProperty` that is exposed on the same class and that is the identifier of the bindable property. For example, the corresponding bindable property identifier for the `Label.Text` property is `Label.TextProperty`.

## Creating and Consuming a Bindable Property

The process for creating a bindable property is as follows:

1. Create a `BindableProperty` instance with one of the `BindableProperty.Create` method overloads.
2. Define property accessors for the `BindableProperty` instance.

Note that all `BindableProperty` instances must be created on the UI thread. This means that only code that runs on the UI thread can get or set the value of a bindable property. However, `BindableProperty` instances can be accessed from other threads by marshaling to the UI thread with the `Device.BeginInvokeOnMainThread` method.

### Creating a Property

To create a `BindableProperty` instance, the containing class must derive from the `BindableObject` class. However, the `BindableObject` class is high in the class hierarchy, so the majority of classes used for user interface functionality support bindable properties.

A bindable property can be created by declaring a `public static readonly` property of type `BindableProperty`. The bindable property should be set to the returned value of one of the `BindableProperty.Create` method overloads. The declaration should be within the body of `BindableObject` derived class, but outside of any member definitions.

At a minimum, an identifier must be specified when creating a `BindableProperty`, along with the following parameters:

- The name of the `BindableProperty`.
- The type of the property.
- The type of the owning object.
- The default value for the property. This ensures that the property always returns a particular default value when it is unset, and it can be different from the default value for the type of the property. The default value will be restored when the `ClearValue` method is called on the bindable property.

The following code shows an example of a bindable property, with an identifier and values for the four required parameters:

```
public static readonly BindableProperty EventNameProperty =
    BindableProperty.Create ("EventName", typeof(string), typeof(EventToCommandBehavior), null);
```

This creates a `BindableProperty` instance named `EventName`, of type `string`. The property is owned by the `EventToCommandBehavior` class, and has a default value of `null`. The naming convention for bindable properties is that the bindable property identifier must match the property name specified in the `Create` method, with "Property" appended to it. Therefore, in the example above, the bindable property identifier is `EventNameProperty`.

Optionally, when creating a `BindableProperty` instance, the following parameters can be specified:

- The binding mode. This is used to specify the direction in which property value changes will propagate. In the default binding mode, changes will propagate from the *source* to the *target*.
- A validation delegate that will be invoked when the property value is set. For more information, see [Validation Callbacks](#).
- A property changed delegate that will be invoked when the property value has changed. For more information, see [Detecting Property Changes](#).
- A property changing delegate that will be invoked when the property value will change. This delegate has the same signature as the property changed delegate.
- A coerce value delegate that will be invoked when the property value has changed. For more information, see [Coerce Value Callbacks](#).
- A `Func` that's used to initialize a default property value. For more information, see [Creating a Default Value with a Func](#).

## Creating Accessors

Property accessors are required to use property syntax to access a bindable property. The `Get` accessor should return the value that's contained in the corresponding bindable property. This can be achieved by calling the `GetValue` method, passing in the bindable property identifier on which to get the value, and then casting the result to the required type. The `Set` accessor should set the value of the corresponding bindable property. This can be achieved by calling the `SetValue` method, passing in the bindable property identifier on which to set the value, and the value to set.

The following code example shows accessors for the `EventName` bindable property:

```
public string EventName {
    get { return (string)GetValue (EventNameProperty); }
    set { SetValue (EventNameProperty, value); }
}
```

## Consuming a Bindable Property

Once a bindable property has been created, it can be consumed from XAML or code. In XAML, this is achieved by declaring a namespace with a prefix, with the namespace declaration indicating the CLR namespace name, and optionally, an assembly name. For more information, see [XAML Namespaces](#).

The following code example demonstrates a XAML namespace for a custom type that contains a bindable property, which is defined within the same assembly as the application code that's referencing the custom type:

```
<ContentPage ... xmlns:local="clr-namespace:EventToCommandBehavior" ...>
...
</ContentPage>
```

The namespace declaration is used when setting the `EventName` bindable property, as demonstrated in the following XAML code example:

```
<ListView ...>
<ListView.Behaviors>
    <local:EventToCommandBehavior EventName="ItemSelected" ... />
</ListView.Behaviors>
</ListView>
```

The equivalent C# code is shown in the following code example:

```
var listView = new ListView ();
listView.Behaviors.Add (new EventToCommandBehavior {
    EventName = "ItemSelected",
    ...
});
```

## Advanced Scenarios

When creating a `BindableProperty` instance, there are a number of optional parameters that can be set to enable advanced bindable property scenarios. This section explores these scenarios.

### Detecting Property Changes

A `static` property-changed callback method can be registered with a bindable property by specifying the `PropertyChanged` parameter for the `BindableProperty.Create` method. The specified callback method will be invoked when the value of the bindable property changes.

The following code example shows how the `EventName` bindable property registers the `OnEventNameChanged` method as a property-changed callback method:

```
public static readonly BindableProperty EventNameProperty =
BindableProperty.Create (
    "EventName", typeof(string), typeof(EventToCommandBehavior), null, PropertyChanged: OnEventNameChanged);
...

static void OnEventNameChanged (BindableObject bindable, object oldValue, object newValue)
{
    // Property changed implementation goes here
}
```

In the property-changed callback method, the `BindableObject` parameter is used to denote which instance of the owning class has reported a change, and the values of the two `object` parameters represent the old and new values of the bindable property.

### Validation Callbacks

A `static` validation callback method can be registered with a bindable property by specifying the `validateValue` parameter for the `BindableProperty.Create` method. The specified callback method will be invoked when the value of the bindable property is set.

The following code example shows how the `Angle` bindable property registers the `IsValidValue` method as a validation callback method:

```
public static readonly BindableProperty AngleProperty =
    BindableProperty.Create ("Angle", typeof(double), typeof(HomePage), 0.0, validateValue: IsValidValue);
...
static bool IsValidValue (BindableObject view, object value)
{
    double result;
    bool isDouble = double.TryParse (value.ToString (), out result);
    return (result >= 0 && result <= 360);
}
```

Validation callbacks are provided with a value, and should return `true` if the value is valid for the property, otherwise `false`. An exception will be raised if a validation callback returns `false`, which should be handled by the developer. A typical use of a validation callback method is constraining the values of integers or doubles when the bindable property is set. For example, the `IsValidValue` method checks that the property value is a `double` within the range 0 to 360.

### Coerce Value Callbacks

A `static` coerce value callback method can be registered with a bindable property by specifying the `coerceValue` parameter for the `BindableProperty.Create` method. The specified callback method will be invoked when the value of the bindable property changes.

Coerce value callbacks are used to force a reevaluation of a bindable property when the value of the property changes. For example, a coerce value callback can be used to ensure that the value of one bindable property is not greater than the value of another bindable property.

The following code example shows how the `Angle` bindable property registers the `CoerceAngle` method as a coerce value callback method:

```
public static readonly BindableProperty AngleProperty = BindableProperty.Create (
    "Angle", typeof(double), typeof(HomePage), 0.0, coerceValue: CoerceAngle);
public static readonly BindableProperty MaximumAngleProperty = BindableProperty.Create (
    "MaximumAngle", typeof(double), typeof(HomePage), 360.0);
...
static object CoerceAngle (BindableObject bindable, object value)
{
    var homePage = bindable as HomePage;
    double input = (double)value;

    if (input > homePage.MaximumAngle) {
        input = homePage.MaximumAngle;
    }

    return input;
}
```

The `CoerceAngle` method checks the value of the `MaximumAngle` property, and if the `Angle` property value is greater than it, it coerces the value to the `MaximumAngle` property value.

### Creating a Default Value with a Func

A `Func` can be used to initialize the default value of a bindable property, as demonstrated in the following code example:

```
public static readonly BindableProperty SizeProperty =
BindableProperty.Create ("Size", typeof(double), typeof(HomePage), 0.0,
defaultValueCreator: bindable => Device.GetNamedSize (NamedSize.Large, (Label)bindable));
```

The `defaultValueCreator` parameter is set to a `Func` that invokes the `Device.GetNamedSize` method to return a `double` that represents the named size for the font that is used on a `Label` on the native platform.

## Summary

This article provided an introduction to bindable properties, and demonstrated how to create and consume them. A bindable property is a special type of property, where the property's value is tracked by the Xamarin.Forms property system.

## Related Links

- [XAML Namespaces](#)
- [Event To Command Behavior \(sample\)](#)
- [Validation Callback \(sample\)](#)
- [Coerce Value Callback \(sample\)](#)
- [BindableProperty](#)
- [BindableObject](#)

# Attached Properties

11/12/2018 • 4 minutes to read • [Edit Online](#)

An attached property is a special type of bindable property, defined in one class but attached to other objects, and recognizable in XAML as an attribute that contains a class and a property name separated by a period. This article provides an introduction to attached properties, and demonstrates how to create and consume them.

## Overview

Attached properties enable an object to assign a value for a property that its own class doesn't define. For example, child elements can use attached properties to inform their parent element of how they are to be presented in the user interface. The `Grid` control allows the row and column of a child to be specified by setting the `Grid.Row` and `Grid.Column` attached properties. `Grid.Row` and `Grid.Column` are attached properties because they are set on elements that are children of a `Grid`, rather than on the `Grid` itself.

Bindable properties should be implemented as attached properties in the following scenarios:

- When there's a need to have a property setting mechanism available for classes other than the defining class.
- When the class represents a service that needs to be easily integrated with other classes.

For more information about bindable properties, see [Bindable Properties](#).

## Creating and Consuming an Attached Property

The process for creating an attached property is as follows:

1. Create a `BindableProperty` instance with one of the `CreateAttached` method overloads.
2. Provide `static Get PropertyName` and `set PropertyName` methods as accessors for the attached property.

### Creating a Property

When creating an attached property for use on other types, the class where the property is created does not have to derive from `BindableObject`. However, the *target* property for accessors should be of, or derive from, `BindableObject`.

An attached property can be created by declaring a `public static readonly` property of type `BindableProperty`. The bindable property should be set to the returned value of one of the `BindableProperty.CreateAttached` method overloads. The declaration should be within the body of the owning class, but outside of any member definitions.

The following code shows an example of an attached property:

```
public static readonly BindableProperty HasShadowProperty =
    BindableProperty.CreateAttached ("HasShadow", typeof(bool), typeof(ShadowEffect), false);
```

This creates an attached property named `HasShadow`, of type `bool`. The property is owned by the `ShadowEffect` class, and has a default value of `false`. The naming convention for attached properties is that the attached property identifier must match the property name specified in the `CreateAttached` method, with "Property" appended to it. Therefore, in the example above, the attached property identifier is `HasShadowProperty`.

For more information about creating bindable properties, including parameters that can be specified during creation, see [Creating and Consuming a Bindable Property](#).

### Creating Accessors

Static `Get` *PropertyName* and `Set` *PropertyName* methods are required as accessors for the attached property, otherwise the property system will be unable to use the attached property. The `Get` *PropertyName* accessor should conform to the following signature:

```
public static valueType GetPropertyName(BindableObject target)
```

The `Get` *PropertyName* accessor should return the value that's contained in the corresponding `BindableProperty` field for the attached property. This can be achieved by calling the `GetValue` method, passing in the bindable property identifier on which to get the value, and then casting the resulting value to the required type.

The `set` *PropertyName* accessor should conform to the following signature:

```
public static void SetPropertyName(BindableObject target, valueType value)
```

The `set` *PropertyName* accessor should set the value of the corresponding `BindableProperty` field for the attached property. This can be achieved by calling the `SetValue` method, passing in the bindable property identifier on which to set the value, and the value to set.

For both accessors, the *target* object should be of, or derive from, `BindableObject`.

The following code example shows accessors for the `HasShadow` attached property:

```
public static bool GetHasShadow (BindableObject view)
{
    return (bool)view.GetValue (HasShadowProperty);
}

public static void SetHasShadow (BindableObject view, bool value)
{
    view.SetValue (HasShadowProperty, value);
}
```

## Consuming an Attached Property

Once an attached property has been created, it can be consumed from XAML or code. In XAML, this is achieved by declaring a namespace with a prefix, with the namespace declaration indicating the Common Language Runtime (CLR) namespace name, and optionally an assembly name. For more information, see [XAML Namespaces](#).

The following code example demonstrates a XAML namespace for a custom type that contains an attached property, which is defined within the same assembly as the application code that's referencing the custom type:

```
<ContentPage ... xmlns:local="clr-namespace:EffectsDemo" ...>
...
</ContentPage>
```

The namespace declaration is then used when setting the attached property on a specific control, as demonstrated in the following XAML code example:

```
<Label Text="Label Shadow Effect" local:ShadowEffect.HasShadow="true" />
```

The equivalent C# code is shown in the following code example:

```
var label = new Label { Text = "Label Shadow Effect" };
ShadowEffect.SetHasShadow (label, true);
```

## Consuming an Attached Property with a Style

Attached properties can also be added to a control by a style. The following XAML code example shows an *explicit* style that uses the `HasShadow` attached property, that can be applied to `Label` controls:

```
<Style x:Key="ShadowEffectStyle" TargetType="Label">
  <Style.Setters>
    <Setter Property="local:ShadowEffect.HasShadow" Value="true" />
  </Style.Setters>
</Style>
```

The `style` can be applied to a `Label` by setting its `Style` property to the `style` instance using the `StaticResource` markup extension, as demonstrated in the following code example:

```
<Label Text="Label Shadow Effect" Style="{StaticResource ShadowEffectStyle}" />
```

For more information about styles, see [Styles](#).

## Advanced Scenarios

When creating an attached property, there are a number of optional parameters that can be set to enable advanced attached property scenarios. This includes detecting property changes, validating property values, and coercing property values. For more information, see [Advanced Scenarios](#).

## Summary

This article provided an introduction to attached properties, and demonstrated how to create and consume them. An attached property is a special type of bindable property, defined in one class but attached to other objects, and recognizable in XAML as attributes that contain a class and a property name separated by a period.

## Related Links

- [Bindable Properties](#)
- [XAML Namespaces](#)
- [Shadow Effect \(sample\)](#)
- [BindableProperty](#)
- [BindableObject](#)

# Resource Dictionaries

11/12/2018 • 8 minutes to read • [Edit Online](#)

*XAML resources are definitions of objects that can be shared and re-used throughout a Xamarin.Forms application.*

These resource objects are stored in a resource dictionary. This article describes how to create and consume a resource dictionary, and how to merge resource dictionaries.

## Overview

A `ResourceDictionary` is a repository for resources that are used by a Xamarin.Forms application. Typical resources that are stored in a `ResourceDictionary` include [styles](#), [control templates](#), [data templates](#), colors, and converters.

In XAML, resources that are stored in a `ResourceDictionary` can then be retrieved and applied to elements by using the `StaticResource` markup extension. In C#, resources can also be defined in a `ResourceDictionary` and then retrieved and applied to elements by using a string-based indexer. However, there's little advantage to using a `ResourceDictionary` in C#, as shared objects can simply be stored as fields or properties, and accessed directly without having to first retrieve them from a dictionary.

## Creating and Consuming a ResourceDictionary

Resources are defined in a `ResourceDictionary` that is then set to one of the following `Resources` properties:

- The `Resources` property of any class that derives from `Application`
- The `Resources` property of any class that derives from `VisualElement`

A Xamarin.Forms program contains only one class that derives from `Application` but often makes use of many classes that derive from `visualElement`, including pages, layouts, and controls. Any of these objects can have its `Resources` property set to a `ResourceDictionary`. Choosing where to put a particular `ResourceDictionary` impacts where the resources can be used:

- Resources in a `ResourceDictionary` that is attached to a view such as `Button` or `Label` can only be applied to that particular object, so this is not very useful.
- Resources in a `ResourceDictionary` attached to a layout such as `StackLayout` or `Grid` can be applied to the layout and all the children of that layout.
- Resources in a `ResourceDictionary` defined at the page level can be applied to the page and to all its children.
- Resources in a `ResourceDictionary` defined at the application level can be applied throughout the application.

The following XAML shows resources defined in an application level `ResourceDictionary` in the `App.xaml` file created as part of the standard Xamarin.Forms program:

```
<Application ...>
  <Application.Resources>
    <ResourceDictionary>
      <Color x:Key="PageBackgroundColor">Yellow</Color>
      <Color x:Key="HeadingTextColor">Black</Color>
      <Color x:Key="NormalTextColor">Blue</Color>
      <Style x:Key="LabelPageHeadingStyle" TargetType="Label">
        <Setter Property="FontAttributes" Value="Bold" />
        <Setter Property="HorizontalOptions" Value="Center" />
        <Setter Property="TextColor" Value="{StaticResource HeadingTextColor}" />
      </Style>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

This `ResourceDictionary` defines three `Color` resources and a `Style` resource. For more information about the `App` class, see [App Class](#).

Beginning in Xamarin.Forms 3.0, the explicit `ResourceDictionary` tags are not required. The `ResourceDictionary` object is created automatically, and you can insert the resources directly between the `Resources` property-element tags:

```
<Application ...>
  <Application.Resources>
    <Color x:Key="PageBackgroundColor">Yellow</Color>
    <Color x:Key="HeadingTextColor">Black</Color>
    <Color x:Key="NormalTextColor">Blue</Color>
    <Style x:Key="LabelPageHeadingStyle" TargetType="Label">
      <Setter Property="FontAttributes" Value="Bold" />
      <Setter Property="HorizontalOptions" Value="Center" />
      <Setter Property="TextColor" Value="{StaticResource HeadingTextColor}" />
    </Style>
  </Application.Resources>
</Application>
```

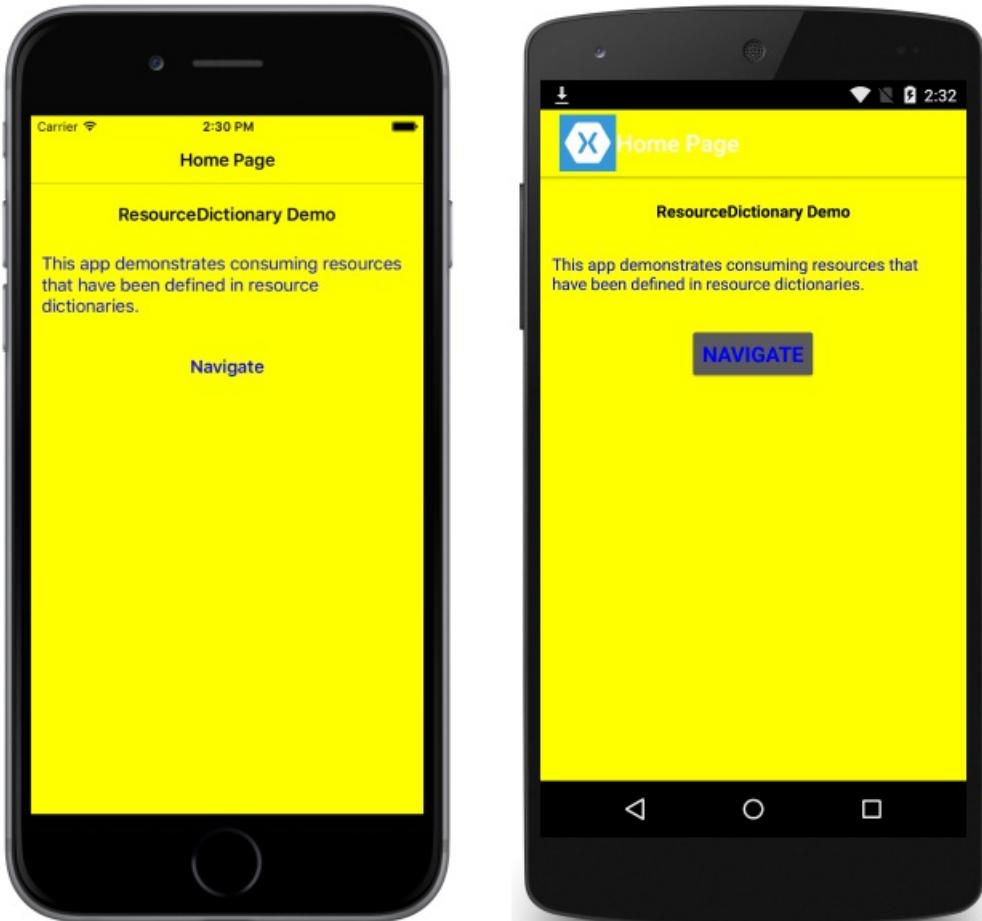
Each resource has a key that is specified using the `x:Key` attribute, which becomes its dictionary key in the `ResourceDictionary`. The key is used to retrieve a resource from the `ResourceDictionary` by the `StaticResource` markup extension, as demonstrated in the following XAML code example that shows additional resources defined within the `StackLayout`:

```

<StackLayout Margin="0,20,0,0">
    <StackLayout.Resources>
        <ResourceDictionary>
            <Style x:Key="LabelNormalStyle" TargetType="Label">
                <Setter Property="TextColor" Value="{StaticResource NormalTextColor}" />
            </Style>
            <Style x:Key="MediumBoldText" TargetType="Button">
                <Setter Property="FontSize" Value="Medium" />
                <Setter Property="FontAttributes" Value="Bold" />
            </Style>
        </ResourceDictionary>
    </StackLayout.Resources>
    <Label Text="ResourceDictionary Demo" Style="{StaticResource LabelPageHeadingStyle}" />
    <Label Text="This app demonstrates consuming resources that have been defined in resource dictionaries." Margin="10,20,10,0" Style="{StaticResource LabelNormalStyle}" />
    <Button Text="Navigate" Clicked="OnNavigateButtonClicked" TextColor="{StaticResource NormalTextColor}" Margin="0,20,0,0" HorizontalOptions="Center" Style="{StaticResource MediumBoldText}" />
</StackLayout>

```

The first `Label` instance retrieves and consumes the `LabelPageHeadingStyle` resource defined in the application level `ResourceDictionary`, with the second `Label` instance retrieving and consuming the `LabelNormalStyle` resource defined in the control level `ResourceDictionary`. Similarly, the `Button` instance retrieves and consumes the `NormalTextColor` resource defined in the application level `ResourceDictionary`, and the `MediumBoldText` resource defined in the control level `ResourceDictionary`. This results in the appearance shown in the following screenshots:



#### NOTE

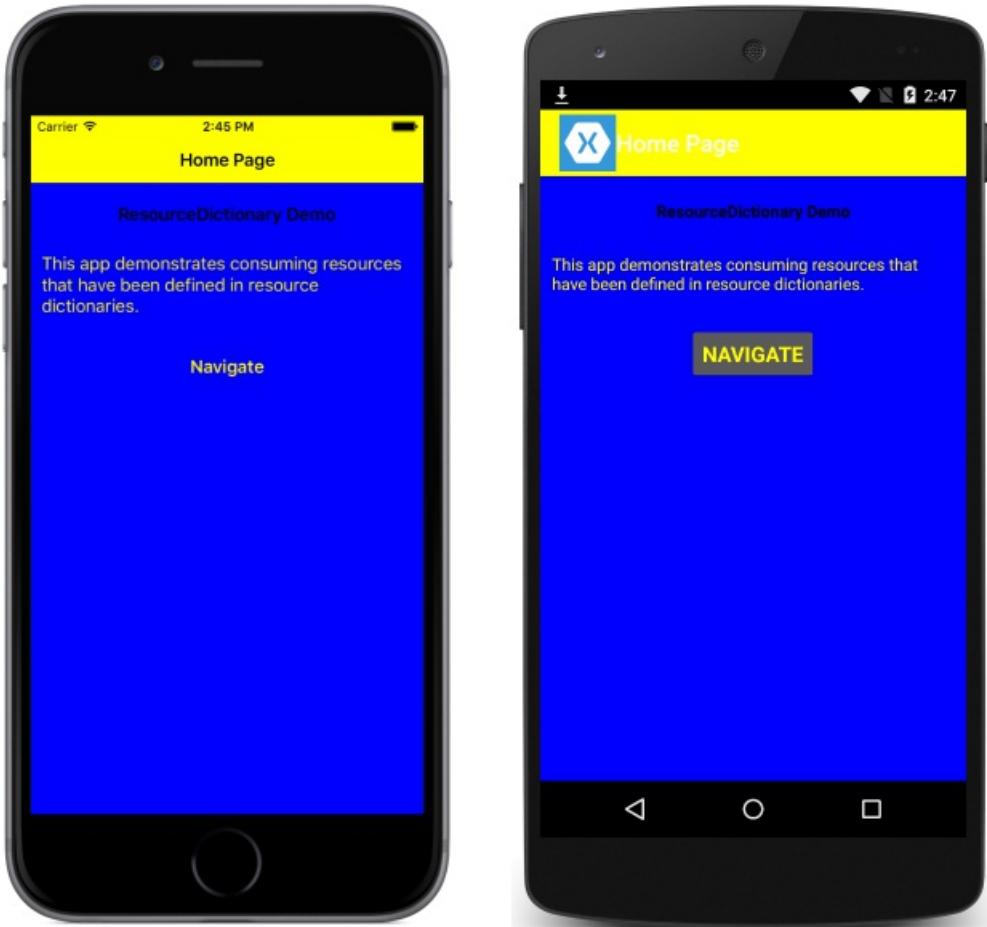
Resources that are specific to a single page shouldn't be included in an application level resource dictionary, as such resources will then be parsed at application startup instead of when required by a page. For more information, see [Reduce the Application Resource Dictionary Size](#).

## Overriding Resources

When `ResourceDictionary` resources share `x:Key` attribute values, resources defined lower in the view hierarchy will take precedence over those defined higher up. For example, setting the `PageBackgroundColor` resource to `Blue` at the application level will be overridden by a page level `PageBackgroundColor` resource set to `Yellow`. Similarly, a page level `PageBackgroundColor` resource will be overridden by a control level `PageBackgroundColor` resource. This precedence is demonstrated by the following XAML code example:

```
<ContentPage ... BackgroundColor="{StaticResource PageBackgroundColor}">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Color x:Key="PageBackgroundColor">Blue</Color>
            <Color x:Key="NormalTextColor">Yellow</Color>
        </ResourceDictionary>
    </ContentPage.Resources>
    <StackLayout Margin="0,20,0,0">
        ...
        <Label Text="ResourceDictionary Demo" Style="{StaticResource LabelPageHeadingStyle}" />
        <Label Text="This app demonstrates consuming resources that have been defined in resource
dictionaries."
            Margin="10,20,10,0"
            Style="{StaticResource LabelNormalStyle}" />
        <Button Text="Navigate"
            Clicked="OnNavigateButtonClicked"
            TextColor="{StaticResource NormalTextColor}"
            Margin="0,20,0,0"
            HorizontalOptions="Center"
            Style="{StaticResource MediumBoldText}" />
    </StackLayout>
</ContentPage>
```

The original `PageBackgroundColor` and `NormalTextColor` instances, defined at the application level, are overridden by the `PageBackgroundColor` and `NormalTextColor` instances defined at page level. Therefore, the page background color becomes blue, and the text on the page becomes yellow, as demonstrated in the following screenshots:



However, note that the background bar of the `NavigationPage` is still yellow, because the `BarBackgroundColor` property is set to the value of the `PageBackgroundColor` resource defined in the application level `ResourceDictionary`.

Here's another way to think about `ResourceDictionary` precedence: When the XAML parser encounters a `StaticResource`, it searches for a matching key by traveling up through the visual tree, using the first match it finds. If this search ends at the page and the key still hasn't been found, the XAML parser searches the `ResourceDictionary` attached to the `App` object. If the key is still not found, an exception is raised.

## Stand-alone Resource Dictionaries

A class derived from `ResourceDictionary` can also be in a separate stand-alone file. (More precisely, a class derived from `ResourceDictionary` generally requires a *pair* of files because the resources are defined in a XAML file but a code-behind file with an `InitializeComponent` call is also necessary.) The resultant file can then be shared among applications.

To create such a file, add a new **Content View** or **Content Page** item to the project (but not a **Content View** or **Content Page** with only a C# file). In both the XAML file and C# file, change the name of the base class from `ContentView` or `ContentPage` to `ResourceDictionary`. In the XAML file, the name of the base class is the top-level element.

The following XAML example shows a `ResourceDictionary` named `MyResourceDictionary`:

```

<ResourceDictionary xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ResourceDictionaryDemo.MyResourceDictionary">
    <DataTemplate x:Key="PersonDataTemplate">
        <ViewCell>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="0.5*" />
                    <ColumnDefinition Width="0.2*" />
                    <ColumnDefinition Width="0.3*" />
                </Grid.ColumnDefinitions>
                <Label Text="{Binding Name}" TextColor="{StaticResource NormalTextColor}"
FontAttributes="Bold" />
                <Label Grid.Column="1" Text="{Binding Age}" TextColor="{StaticResource NormalTextColor}" />
                <Label Grid.Column="2" Text="{Binding Location}" TextColor="{StaticResource NormalTextColor}" />
            </Grid>
        </ViewCell>
    </DataTemplate>
</ResourceDictionary>

```

This `ResourceDictionary` contains a single resource, which is an object of type `DataTemplate`.

You can instantiate `MyResourceDictionary` by putting it between a pair of `Resources` property-element tags, for example, in a `ContentPage`:

```

<ContentPage ...>
    <ContentPage.Resources>
        <local:MyResourceDictionary />
    </ContentPage.Resources>
    ...
</ContentPage>

```

An instance of `MyResourceDictionary` is set to the `Resources` property of the `ContentPage` object.

However, this approach has some limitations: The `Resources` property of the `ContentPage` references only this one `ResourceDictionary`. In most cases, you want the option of including other `ResourceDictionary` instances and perhaps other resources as well.

This task requires merged resource dictionaries.

## Merged Resource Dictionaries

Merged resource dictionaries combine one or more `ResourceDictionary` instances into another `ResourceDictionary`. You can do this in a XAML file by setting the `MergedDictionaries` property to one or more resource dictionaries that will be merged into the application, page, or control level `ResourceDictionary`.

### IMPORTANT

`ResourceDictionary` also defines a `MergedWith` property. Do not use this property; it has been deprecated as of Xamarin.Forms 3.0.

An instance of `MyResourceDictionary` can be merged into any application, page, or control level `ResourceDictionary`. The following XAML code example shows it being merged into a page level `ResourceDictionary` using the `MergedDictionaries` property:

```

<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <local:MyResourceDictionary />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </ContentPage.Resources>
  ...
</ContentPage>

```

That markup shows only an instance of `MyResourceDictionary` being added to the `ResourceDictionary` but you can also reference other `ResourceDictionary` instances within the `MergedDictionary` property-element tags, and other resources outside of those tags:

```

<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary>

      <!-- Add more resources here -->

      <ResourceDictionary.MergedDictionaries>

        <!-- Add more resource dictionaries here -->

        <local:MyResourceDictionary />

        <!-- Add more resource dictionaries here -->

      </ResourceDictionary.MergedDictionaries>

      <!-- Add more resources here -->

    </ResourceDictionary>
  </ContentPage.Resources>
  ...
</ContentPage>

```

There can be only one `MergedDictionaries` section in a `ResourceDictionary`, but you can put as many `ResourceDictionary` instances in there as you want.

When merged `ResourceDictionary` resources share identical `x:Key` attribute values, Xamarin.Forms uses the following resource precedence:

1. The resources local to the resource dictionary.
2. The resources contained in the resource dictionary that was merged via the deprecated `MergedWith` property.
3. The resources contained in the resource dictionaries that were merged via the `MergedDictionaries` collection, in the reverse order they are listed in the `MergedDictionaries` property.

#### **NOTE**

Searching resource dictionaries can be a computationally intensive task if an application contains multiple, large resource dictionaries. Therefore, to avoid unnecessary searching, you should ensure that each page in an application only uses resource dictionaries that are appropriate to the page.

## Merging Dictionaries in Xamarin.Forms 3.0

Beginning with Xamarin.Forms 3.0, the process of merging `ResourceDictionary` instances has become somewhat

easier and more flexible. The `MergedDictionaries` property-element tags are no longer necessary. Instead, you add to the resource dictionary another `ResourceDictionary` tag with the new `Source` property set to the filename of the XAML file with the resources:

```
<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary>

      <!-- Add more resources here -->

      <ResourceDictionary Source="MyResourceDictionary.xaml" />

      <!-- Add more resources here -->

    </ResourceDictionary>
  </ContentPage.Resources>
  ...
</ContentPage>
```

Because Xamarin.Forms 3.0 automatically instantiates the `ResourceDictionary`, those two outer `ResourceDictionary` tags are no longer required:

```
<ContentPage ...>
  <ContentPage.Resources>

    <!-- Add more resources here -->

    <ResourceDictionary Source="MyResourceDictionary.xaml" />

    <!-- Add more resources here -->

  </ContentPage.Resources>
  ...
</ContentPage>
```

This new syntax does *not* instantiate the `MyResourceDictionary` class. Instead, it references the XAML file. For that reason the code-behind file (**MyResourceDictionary.xaml.cs**) is no longer required. You can also remove the `x:Class` attribute from the root tag of the **MyResourceDictionary.xaml** file.

## Summary

This article explained how to create and consume a `ResourceDictionary`, and how to merge resource dictionaries. A `ResourceDictionary` allows resources to be defined in a single location, and re-used throughout a Xamarin.Forms application.

## Related Links

- [Resource Dictionaries \(sample\)](#)
- [Styles](#)
- [ResourceDictionary](#)

# XAML Standard (Preview)

11/12/2018 • 2 minutes to read • [Edit Online](#)



Follow these steps to experiment with XAML Standard in Xamarin.Forms:

- [Visual Studio](#)
- [Visual Studio for Mac](#)

1. Download the preview NuGet package [here](#).
2. Add the **Xamarin.Forms.Alias** NuGet package to your Xamarin.Forms .NET Standard and platform projects.
3. Initialize the package with `Alias.Init()`
4. Add an `xmlns:a` reference `xmlns:a="clr-namespace:Xamarin.Forms.Alias;assembly=Xamarin.Forms.Alias"`
5. Use the types in XAML - see the [Controls reference](#) for more information.

The following XAML demonstrates some XAML Standard controls being used in a Xamarin.Forms `ContentPage`:

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:a="clr-namespace:Xamarin.Forms.Alias;assembly=Xamarin.Forms.Alias"
    x:Class="XAMLStandardSample.ItemsPage"
    Title="{Binding Title}" x:Name="BrowseItemsPage">
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add" Clicked="AddItem_Clicked" />
    </ContentPage.ToolbarItems>
    <ContentPage.Content>
        <a:StackPanel>
            <ListView x:Name="ItemsListView" ItemsSource="{Binding Items}" VerticalOptions="FillAndExpand"
                HasUnevenRows="true" RefreshCommand="{Binding LoadItemsCommand}" IsPullToRefreshEnabled="true" IsRefreshing="
                {Binding IsBusy, Mode=OneWay}" CachingStrategy="RecycleElement" ItemSelected="OnItemSelected">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <StackLayout Padding="10">
                                <a:TextBlock Text="{Binding Text}" LineBreakMode="NoWrap" Style="{DynamicResource
ListTextStyle}" FontSize="16" />
                                <a:TextBlock Text="{Binding Description}" LineBreakMode="NoWrap" Style="{DynamicResource
ListDetailTextStyle}" FontSize="13" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </a:StackPanel>
    </ContentPage.Content>
</ContentPage>
```

**NOTE**

Requiring the xmlns `a:` prefix on the XAML Standard controls is a limitation of the current preview.

## Related Links

- [Preview NuGet](#)
- [Controls Reference](#)

# XAML Standard (Preview) Controls

11/12/2018 • 2 minutes to read • [Edit Online](#)



This page lists the XAML Standard controls available in the Preview, alongside their equivalent Xamarin.Forms control.

There is also a list of controls that have new property and enumeration names in XAML Standard.

## Controls

XAMARIN.FORMS	XAML STANDARD
Frame	Border
Picker	ComboBox
ActivityIndicator	ProgressRing
StackLayout	StackPanel
Label	TextBlock
Entry	TextBox
Switch	ToggleSwitch
ContentView	UserControl

## Properties and Enumerations

XAMARIN.FORMS CONTROLS WITH UPDATED PROPERTIES	XAMARIN.FORMS PROPERTY OR ENUM	XAML STANDARD EQUIVALENT
Button, Entry, Label, DatePicker, Editor, SearchBar, TimePicker	TextColor	Foreground
VisualElement	BackgroundColor	Background *
Picker, Button	BorderColor, OutlineColor	BorderBrush
Button	BorderWidth	BorderThickness
ProgressBar	Progress	Value

XAMARIN.FORMS CONTROLS WITH UPDATED PROPERTIES	XAMARIN.FORMS PROPERTY OR ENUM	XAML STANDARD EQUIVALENT
Button, Entry, Label, Editor, SearchBar, Span, Font	FontAttributesBold, Italic, None	FontStyleItalic, Normal
Button, Entry, Label, Editor, SearchBar, Span, Font	FontAttributes	FontWeights *Bold, Normal
InputView	KeyboardDefault, Url, Number, Telephone, Text, Chat, Email	InputScopeNameValue *Default, Url, Number, TelephoneNumber, Text, Chat, EmailNameOrAddress
StackPanel	StackOrientation	Orientation *

#### IMPORTANT

Items marked with \* are incomplete in the current preview

## Related Links

- [Preview NuGet](#)

# Xamarin.Forms Application Fundamentals

7/12/2018 • 2 minutes to read • [Edit Online](#)

## Accessibility

Tips to incorporate accessible features (like supporting screen-reading tools) with Xamarin.Forms.

## App Class

The `Application` class is the starting point for Xamarin.Forms – every app needs to implement a subclass `App` to set the initial page. It also provides the `Properties` collection for simple data storage. It can be defined in either C# or XAML.

## App Lifecycle

The `Application` class `OnStart`, `OnSleep`, and `OnResume` methods, as well as modal navigation events, let you handle application lifecycle events with custom code.

## Behaviors

User interface controls can be easily extended without subclassing by using behaviors to add functionality.

## Custom Renderers

Custom Renders let developers 'override' the default rendering of Xamarin.Forms controls to customize their appearance and behavior on each platform (using native SDKs if desired).

## Data Binding

Data binding links the properties of two objects, allowing changes in one property to be automatically reflected in the other property. Data binding is an integral part of the Model-View-ViewModel ([MVVM](#)) application architecture.

## Dependency Service

The `DependencyService` provides a simple locator so that you can code to interfaces in your shared code and provide platform-specific implementations that are automatically resolved, making it easy to reference platform-specific functionality in Xamarin.Forms.

## Effects

Effects allow the native controls on each platform to be customized, and are typically used for small styling changes.

## Files

File handling with Xamarin.Forms can be achieved using code in a .NET Standard library, or by using embedded resources.

## Gestures

The Xamarin.Forms `GestureRecognizer` class supports tap, pinch, and pan gestures on user interface controls.

## Localization

The built-in .NET localization framework can be used to build cross-platform multilingual applications with Xamarin.Forms.

## Local Databases

Xamarin.Forms supports database-driven applications using the SQLite database engine, which makes it possible to load and save objects in shared code.

## Messaging Center

Xamarin.Forms `MessagingCenter` enables view models and other components to communicate with without having to know anything about each other besides a simple Message contract.

## Navigation

Xamarin.Forms provides a number of different page navigation experiences, depending upon the `Page` type being used.

## Templates

Control templates provide the ability to easily theme and re-theme application pages at runtime, while data templates provide the ability to define the presentation of data on supported controls.

## Triggers

Update controls by responding to property changes and events in XAML.

## Related Links

- [Introduction To Xamarin.Forms](#)

# Xamarin.Forms Accessibility

10/23/2018 • 2 minutes to read • [Edit Online](#)

*Building an accessible application ensures that the application is usable by people who approach the user interface with a range of needs and experiences.*

Making a Xamarin.Forms application accessible means thinking about the layout and design of many user interface elements. For guidelines on issues to consider, see the [Accessibility Checklist](#). Many accessibility concerns such as large fonts, and suitable color and contrast settings can already be addressed by Xamarin.Forms APIs.

The [Android accessibility](#) and [iOS accessibility](#) guides contain details of the native APIs exposed by Xamarin, and the [UWP accessibility guide on MSDN](#) explains the native approach on that platform. These APIs are used to fully implement accessible applications on each platform.

Xamarin.Forms does not currently have *built-in* support for all of the accessibility APIs available on each of the underlying platforms. However, it does support setting automation properties on user interface elements to support screen reader and navigation assistance tools, which is one of the most important parts of building accessible applications. For more information, see [Automation Properties](#).

Xamarin.Forms applications can also have the tab order of controls specified. For more information, see [Keyboard Navigation](#).

Other accessibility APIs (such as [PostNotification on iOS](#)) may be better suited to a [DependencyService](#) or [Custom Renderer](#) implementation. These are not covered in this guide.

## Testing Accessibility

Xamarin.Forms applications typically target multiple platforms, which means testing the accessibility features according to the platform. Follow these links to learn how to test accessibility on each platform:

- [iOS Testing](#)
- [Android Testing](#)
- [Windows AccScope \(MSDN\)](#)

## Related Links

- [Cross-platform Accessibility](#)
- [Automation Properties](#)
- [Keyboard Accessibility](#)

# Automation Properties in Xamarin.Forms

10/18/2018 • 4 minutes to read • [Edit Online](#)

Xamarin.Forms allows accessibility values to be set on user interface elements by using attached properties from the `AutomationProperties` class, which in turn set native accessibility values. This article explains how to use the `AutomationProperties` class, so that a screen reader can speak about the elements on the page.

Xamarin.Forms allows automation properties to be set on user interface elements via the following attached properties:

- `AutomationProperties.IsInAccessibleTree` – indicates whether the element is available to an accessible application. For more information, see [AutomationProperties.IsInAccessibleTree](#).
- `AutomationProperties.Name` – a short description of the element that serves as a speakable identifier for the element. For more information, see [AutomationProperties.Name](#).
- `AutomationProperties.HelpText` – a longer description of the element, which can be thought of as tooltip text associated with the element. For more information, see [AutomationProperties.HelpText](#).
- `AutomationProperties.LabeledBy` – allows another element to define accessibility information for the current element. For more information, see [AutomationProperties.LabeledBy](#).

These attached properties set native accessibility values so that a screen reader can speak about the element. For more information about attached properties, see [Attached Properties](#).

## IMPORTANT

Using the `AutomationProperties` attached properties may impact UI Test execution on Android. The `AutomationId`, `AutomationProperties.Name` and `AutomationProperties.HelpText` properties will both set the native `ContentDescription` property, with the `AutomationProperties.Name` and `AutomationProperties.HelpText` property values taking precedence over the `AutomationId` value (if both `AutomationProperties.Name` and `AutomationProperties.HelpText` are set, the values will be concatenated). This means that any tests looking for `AutomationId` will fail if `AutomationProperties.Name` or `AutomationProperties.HelpText` are also set on the element. In this scenario, UI Tests should be altered to look for the value of `AutomationProperties.Name` or `AutomationProperties.HelpText`, or a concatenation of both.

Each platform has a different screen reader to narrate the accessibility values:

- iOS has VoiceOver. For more information, see [Test Accessibility on Your Device with VoiceOver](#) on developer.apple.com.
- Android has TalkBack. For more information, see [Testing Your App's Accessibility](#) on developer.android.com.
- Windows has Narrator. For more information, see [Verify main app scenarios by using Narrator](#).

However, the exact behavior of a screen reader depends on the software and on the user's configuration of it. For example, most screen readers read the text associated with a control when it receives focus, enabling users to orient themselves as they move among the controls on the page. Some screen readers also read the entire application user interface when a page appears, which enables the user to receive all of the page's available informational content before attempting to navigate it.

Screen readers also read different accessibility values. In the sample application:

- VoiceOver will read the `Placeholder` value of the `Entry`, followed by instructions for using the control.
- TalkBack will read the `Placeholder` value of the `Entry`, followed by the `AutomationProperties.HelpText` value,

followed by instructions for using the control.

- Narrator will read the `AutomationProperties.LabeledBy` value of the `Entry`, followed by instructions on using the control.

In addition, Narrator will prioritize `AutomationProperties.Name`, `AutomationProperties.LabeledBy`, and then `AutomationProperties.HelpText`. On Android, TalkBack may combine the `AutomationProperties.Name` and `AutomationProperties.HelpText` values. Therefore, it's recommended that thorough accessibility testing is carried out on each platform to ensure an optimal experience.

## AutomationProperties.IsInAccessibleTree

The `AutomationProperties.IsInAccessibleTree` attached property is a `boolean` that determines if the element is accessible, and hence visible, to screen readers. It must be set to `true` to use the other accessibility attached properties. This can be accomplished in XAML as follows:

```
<Entry AutomationProperties.IsInAccessibleTree="true" />
```

Alternatively, it can be set in C# as follows:

```
var entry = new Entry();
AutomationProperties.SetIsInAccessibleTree(entry, true);
```

### NOTE

Note that the `SetValue` method can also be used to set the `AutomationProperties.IsInAccessibleTree` attached property – `entry.SetValue(AutomationProperties.IsInAccessibleTreeProperty, true);`

## AutomationProperties.Name

The `AutomationProperties.Name` attached property value should be a short, descriptive text string that a screen reader uses to announce an element. This property should be set for elements that have a meaning that is important for understanding the content or interacting with the user interface. This can be accomplished in XAML as follows:

```
<ActivityIndicator AutomationProperties.IsInAccessibleTree="true"
    AutomationProperties.Name="Progress indicator" />
```

Alternatively, it can be set in C# as follows:

```
var activityIndicator = new ActivityIndicator();
AutomationProperties.SetIsInAccessibleTree(activityIndicator, true);
AutomationPropertiesSetName(activityIndicator, "Progress indicator");
```

### NOTE

Note that the `SetValue` method can also be used to set the `AutomationProperties.Name` attached property – `activityIndicator.SetValue(AutomationProperties.NameProperty, "Progress indicator");`

## AutomationProperties.HelpText

The `AutomationProperties.HelpText` attached property should be set to text that describes the user interface element, and can be thought of as tooltip text associated with the element. This can be accomplished in XAML as follows:

```
<Button Text="Toggle ActivityIndicator"  
       AutomationProperties.IsInAccessibleTree="true"  
       AutomationProperties.HelpText="Tap to toggle the activity indicator" />
```

Alternatively, it can be set in C# as follows:

```
var button = new Button { Text = "Toggle ActivityIndicator" };  
AutomationProperties.SetIsInAccessibleTree(button, true);  
AutomationProperties.SetHelpText(button, "Tap to toggle the activity indicator");
```

#### NOTE

Note that the `SetValue` method can also be used to set the `AutomationProperties.HelpText` attached property – `button.SetValue(AutomationProperties.HelpTextProperty, "Tap to toggle the activity indicator");`

On some platforms, for edit controls such as an `Entry`, the `HelpText` property can sometimes be omitted and replaced with placeholder text. For example, "Enter your name here" is a good candidate for the `Entry.Placeholder` property that places the text in the control prior to the user's actual input.

## AutomationProperties.LabeledBy

The `AutomationProperties.LabeledBy` attached property allows another element to define accessibility information for the current element. For example, a `Label` next to an `Entry` can be used to describe what the `Entry` represents. This can be accomplished in XAML as follows:

```
<Label x:Name="label" Text="Enter your name: " />  
<Entry AutomationProperties.IsInAccessibleTree="true"  
      AutomationProperties.LabeledBy="{x:Reference label}" />
```

Alternatively, it can be set in C# as follows:

```
var nameLabel = new Label { Text = "Enter your name: " };  
var entry = new Entry();  
AutomationProperties.SetIsInAccessibleTree(entry, true);  
AutomationProperties.SetLabeledBy(entry, nameLabel);
```

#### NOTE

Note that the `SetValue` method can also be used to set the `AutomationProperties.IsInAccessibleTree` attached property – `entry.SetValue(AutomationProperties.LabeledByProperty, nameLabel);`

## Related Links

- [Attached Properties](#)
- [Accessibility \(sample\)](#)

# Keyboard Navigation in Xamarin.Forms

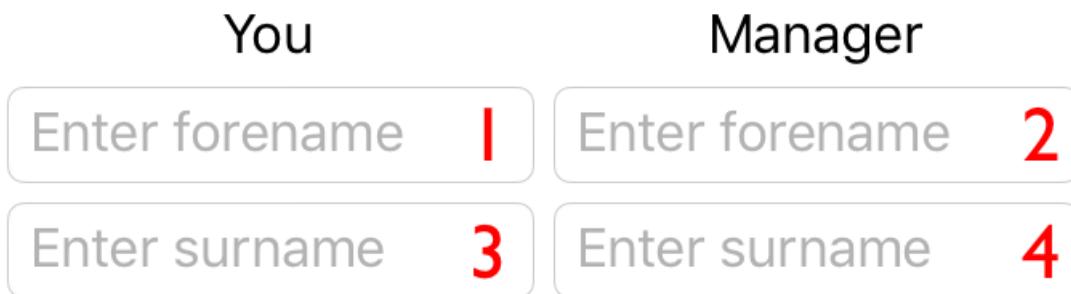
10/23/2018 • 3 minutes to read • [Edit Online](#)

Some users can have difficulty using applications that don't provide appropriate keyboard access. Specifying a tab order for controls enables keyboard navigation and prepares application pages to receive input in a particular order.

By default, the tab order of controls is the same order in which they are listed in XAML, or programmatically added to a child collection. This order is the order in which the controls will be navigated through with a keyboard, and often this default order is the best order. However, the default order is not always the same as the expected order, as shown in the following XAML code example:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.5*" />
        <ColumnDefinition Width="0.5*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Label Text="You"
        HorizontalOptions="Center" />
    <Label Grid.Column="1"
        Text="Manager"
        HorizontalOptions="Center" />
    <Entry Grid.Row="1"
        Placeholder="Enter forename" />
    <Entry Grid.Column="1"
        Grid.Row="1"
        Placeholder="Enter forename" />
    <Entry Grid.Row="2"
        Placeholder="Enter surname" />
    <Entry Grid.Column="1"
        Grid.Row="2"
        Placeholder="Enter surname" />
</Grid>
```

The following screenshot shows the default tab order for this code example:



The tab order here is row-based, and is the order the controls are listed in the XAML. Therefore, pressing the Tab key navigates through forename `Entry` instances, followed by surname `Entry` instances. However, a more intuitive experience would be to use a column-first tab navigation, so that pressing the Tab key navigates through forename-surname pairs. This can be achieved by specifying the tab order of the input controls.

## NOTE

On the Universal Windows Platform, keyboard shortcuts can be defined that provide an intuitive way for users to quickly navigate and interact with the application's visible UI through a keyboard instead of via touch or a mouse. For more information, see [Setting VisualElement Access Keys](#).

## Setting the tab order

The `VisualElement.TabIndex` property is used to indicate the order in which `visualElement` instances receive focus when the user navigates through controls by pressing the Tab key. The default value of the property is 0, and it can be set to any `int` value.

The following rules apply when using the default tab order, or setting the `TabIndex` property:

- `VisualElement` instances with a `TabIndex` equal to 0 are added to the tab order based on their declaration order in XAML or child collections.
- `VisualElement` instances with a `TabIndex` greater than 0 are added to the tab order based on their `TabIndex` value.
- `VisualElement` instances with a `TabIndex` less than 0 are added to the tab order and appear before any zero value.
- Conflicts on a `TabIndex` are resolved by declaration order.

After defining a tab order, pressing the Tab key will cycle the focus through controls in ascending `TabIndex` order, wrapping around to the beginning once the final control is reached.

The following XAML example shows the `TabIndex` property set on input controls to enable column-first tab navigation:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.5*" />
        <ColumnDefinition Width="0.5*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Label Text="You"
        HorizontalOptions="Center" />
    <Label Grid.Column="1"
        Text="Manager"
        HorizontalOptions="Center" />
    <Entry Grid.Row="1"
        Placeholder="Enter forename"
        TabIndex="1" />
    <Entry Grid.Column="1"
        Grid.Row="1"
        Placeholder="Enter forename"
        TabIndex="3" />
    <Entry Grid.Row="2"
        Placeholder="Enter surname"
        TabIndex="2" />
    <Entry Grid.Column="1"
        Grid.Row="2"
        Placeholder="Enter surname"
        TabIndex="4" />
</Grid>
```

The following screenshot shows the tab order for this code example:



The tab order here is column-based. Therefore, pressing the Tab key navigates through forename-surname [Entry](#) pairs.

## Excluding controls from the tab order

In addition to setting the tab order of controls, it may be necessary to exclude controls from the tab order. One way of achieving this is by setting the [IsEnabled](#) property of controls to [false](#), because disabled controls are excluded from the tab order.

However, it may be necessary to exclude controls from the tab order even when they aren't disabled. This can be achieved with the [VisualElement.IsTapStop](#) property, which indicates whether a [VisualElement](#) is included in tab navigation. Its default value is [true](#), and when its value is [false](#) the control is ignored by the tab-navigation infrastructure, irrespective if a [TabIndex](#) is set.

## Supported controls

The [TabIndex](#) and [IsTapStop](#) properties are supported on the following controls, which accept keyboard input on one or more platforms:

- [Button](#)
- [DatePicker](#)
- [Editor](#)
- [Entry](#)
- [NavigationPage](#)
- [Picker](#)
- [ProgressBar](#)
- [SearchBar](#)
- [Slider](#)
- [Stepper](#)
- [Switch](#)
- [TabbedPage](#)
- [TimePicker](#)

### NOTE

Each of these controls isn't tab focusable on every platform.

## Related Links

- [Accessibility \(sample\)](#)



# Xamarin.Forms App Class

10/31/2018 • 4 minutes to read • [Edit Online](#)

The `Application` base class offers the following features, which are exposed in your projects default `App` subclass:

- A `MainPage` property, which is where to set the initial page for the app.
- A persistent `Properties` dictionary to store simple values across lifecycle state changes.
- A static `Current` property that contains a reference to the current application object.

It also exposes [Lifecycle methods](#) such as `OnStart`, `OnSleep`, and `OnResume` as well as modal navigation events.

Depending on which template you chose, the `App` class could be defined in one of two ways:

- **C#**, or
- **XAML & C#**

To create an `App` class using XAML, the default `App` class must be replaced with a XAML `App` class and associated code-behind, as shown in the following code example:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Photos.App">

</Application>
```

The following code example shows the associated code-behind:

```
public partial class App : Application
{
    public App ()
    {
        InitializeComponent ();
        MainPage = new HomePage ();
    }
    ...
}
```

As well as setting the `MainPage` property, the code-behind must also call the `InitializeComponent` method to load and parse the associated XAML.

## MainPage Property

The `MainPage` property on the `Application` class sets the root page of the application.

For example, you can create logic in your `App` class to display a different page depending on whether the user is logged in or not.

The `MainPage` property should be set in the `App` constructor,

```
public class App : Xamarin.Forms.Application
{
    public App ()
    {
        MainPage = new ContentPage { Title = "App Lifecycle Sample" }; // your page here
    }
}
```

## Properties Dictionary

The `Application` subclass has a static `Properties` dictionary which can be used to store data, in particular for use in the `OnStart`, `OnSleep`, and `OnResume` methods. This can be accessed from anywhere in your `Xamarin.Forms` code using `Application.Current.Properties`.

The `Properties` dictionary uses a `string` key and stores an `object` value.

For example, you could set a persistent `"id"` property anywhere in your code (when an item is selected, in a page's `OnDisappearing` method, or in the `OnSleep` method) like this:

```
Application.Current.Properties ["id"] = someClass.ID;
```

In the `OnStart` or `OnResume` methods you can then use this value to recreate the user's experience in some way.

The `Properties` dictionary stores `object`s so you need to cast its value before using it.

```
if (Application.Current.Properties.ContainsKey("id"))
{
    var id = Application.Current.Properties ["id"] as int;
    // do something with id
}
```

Always check for the presence of the key before accessing it to prevent unexpected errors.

### NOTE

The `Properties` dictionary can only serialize primitive types for storage. Attempting to store other types (such as `List<string>`) can fail silently.

## Persistence

The `Properties` dictionary is saved to the device automatically. Data added to the dictionary will be available when the application returns from the background or even after it is restarted.

`Xamarin.Forms` 1.4 introduced an additional method on the `Application` class - `SavePropertiesAsync()` - which can be called to proactively persist the `Properties` dictionary. This is to allow you to save properties after important updates rather than risk them not getting serialized out due to a crash or being killed by the OS.

You can find references to using the `Properties` dictionary in the **Creating Mobile Apps with Xamarin.Forms** book chapters [6](#), [15](#), and [20](#), and in the associated [samples](#).

## The Application Class

A complete `Application` class implementation is shown below for reference:

```

public class App : Xamarin.Forms.Application
{
    public App ()
    {
        MainPage = new ContentPage { Title = "App Lifecycle Sample" }; // your page here
    }

    protected override void OnStart()
    {
        // Handle when your app starts
        Debug.WriteLine ("OnStart");
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
        Debug.WriteLine ("OnSleep");
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
        Debug.WriteLine ("OnResume");
    }
}

```

This class is then instantiated in each platform-specific project and passed to the `LoadApplication` method which is where the `MainPage` is loaded and displayed to the user. The code for each platform is shown in the following sections. The latest Xamarin.Forms solution templates already contain all this code, preconfigured for your app.

## iOS Project

The iOS `AppDelegate` class inherits from `FormsApplicationDelegate`. It should:

- Call `LoadApplication` with an instance of the `App` class.
- Always return `base.FinishedLaunching (app, options);`.

```

[Register ("AppDelegate")]
public partial class AppDelegate :
    global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate // superclass new in 1.3
{
    public override bool FinishedLaunching (UIApplication app, NSDictionary options)
    {
        global::Xamarin.Forms.Forms.Init ();

        LoadApplication (new App ()); // method is new in 1.3

        return base.FinishedLaunching (app, options);
    }
}

```

## Android Project

The Android `MainActivity` inherits from `FormsAppCompatActivity`. In the `OnCreate` override the `LoadApplication` method is called with an instance of the `App` class.

```
[Activity (Label = "App Lifecycle Sample", Icon = "@drawable/icon", Theme = "@style/MainTheme", MainLauncher = true,
    ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
public class MainActivity : FormsAppCompatActivity
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        global::Xamarin.Forms.Forms.Init (this, bundle);

        LoadApplication (new App ()); // method is new in 1.3
    }
}
```

## Universal Windows Project (UWP) for Windows 10

See [Setup Windows Projects](#) for information about UWP support in Xamarin.Forms.

The main page in the UWP project should inherit from `WindowsPage`:

```
<forms:WindowsPage
    ...
    xmlns:forms="using:Xamarin.Forms.Platform.UWP"
    ...>
</forms:WindowsPage>
```

The C# code behind construction must call `LoadApplication` to create an instance of your `Xamarin.Forms App`.

Note that it is good practice to explicitly use the application namespace to qualify the `App` because UWP applications also have their own `App` class unrelated to `Xamarin.Forms`.

```
public sealed partial class MainPage
{
    public MainPage()
    {
        InitializeComponent();

        LoadApplication(new YOUR_NAMESPACE.App());
    }
}
```

Note that `Forms.Init()` must be called in **App.xaml.cs** around line 63.

# Xamarin.Forms App Lifecycle

10/31/2018 • 2 minutes to read • [Edit Online](#)

The `Application` base class offers the following features:

- **Lifecycle methods** `OnStart`, `OnSleep`, and `OnResume`.
- **Page navigation events** `PageAppearing`, `PageDisappearing`.
- **Modal navigation events** `ModalPushing`, `ModalPushed`, `ModalPopping`, and `ModalPopped`.

## Lifecycle Methods

The `Application` class contains three virtual methods that can be overridden to handle lifecycle methods:

- **OnStart** - Called when the application starts.
- **OnSleep** - Called each time the application goes to the background.
- **OnResume** - Called when the application is resumed, after being sent to the background.

Note that there is *no* method for application termination. Under normal circumstances (i.e. not a crash) application termination will happen from the `OnSleep` state, without any additional notifications to your code.

To observe when these methods are called, implement a `.WriteLine` call in each (as shown below) and test on each platform.

```
protected override void OnStart()
{
    Debug.WriteLine ("OnStart");
}
protected override void OnSleep()
{
    Debug.WriteLine ("OnSleep");
}
protected override void OnResume()
{
    Debug.WriteLine ("OnResume");
}
```

When updating *older* Xamarin.Forms applications (eg. create with Xamarin.Forms 1.3 or older), ensure that the Android main activity includes `ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation` in the `[Activity()]` attribute. If this is not present you will observe the `OnStart` method gets called on rotation as well as when the application first starts. This attribute is automatically included in the current Xamarin.Forms app templates.

## Page Navigation events

There are two events on the `Application` class that provide notification of pages appearing and disappearing:

- `PageAppearing` - raised when a page is about to appear on the screen.
- `PageDisappearing` - raised when a page is about to disappear from the screen.

These events can be used in scenarios where you want to track pages as they are appearing on screen.

#### NOTE

The `PageAppearing` and `PageDisappearing` events are raised from the `Page` base class immediately after the `Page.Appearing` and `Page.Disappearing` events, respectively.

## Modal Navigation Events

There are four events on the `Application` class, each with their own event arguments, that let you respond to modal pages being shown and dismissed:

- **ModalPushing** - `ModalPushingEventArgs`
- **ModalPushed** - `ModalPushedEventArgs`
- **ModalPopping** - the `ModalPoppingEventArgs` class contains a `Cancel` property. When `Cancel` is set to `true` the modal pop is cancelled.
- **ModalPopped** - `ModalPoppedEventArgs`

#### NOTE

To implement the application lifecycle methods and modal navigation events, all pre-`Application` methods of creating a Xamarin.Forms app (i.e. applications written in version 1.2 or older that use a static `GetMainPage` method) have been updated to create a default `Application` which is set as the parent of `MainPage`.

Xamarin.Forms applications that use this legacy behavior must be updated to an `Application` subclass as described on the [Application class](#) page.

# Xamarin.Forms Behaviors

7/12/2018 • 2 minutes to read • [Edit Online](#)

*Behaviors lets you add functionality to user interface controls without having to subclass them. Behaviors are written in code and added to controls in XAML or code.*

## Introduction to Behaviors

Behaviors enable you to implement code that you would normally have to write as code-behind, because it directly interacts with the API of the control in such a way that it can be concisely attached to the control. This article provides an introduction to behaviors.

## Attached Behaviors

Attached behaviors are `static` classes with one or more attached properties. This article demonstrates how to create and consume attached behaviors.

## Xamarin.Forms Behaviors

Xamarin.Forms behaviors are created by deriving from the `Behavior` or `Behavior<T>` class. This article demonstrates how to create and consume Xamarin.Forms behaviors.

## Reusable Behaviors

Behaviors are reusable across more than one application. These articles explain how to create useful behaviors to perform commonly used functionality.

# Introduction to Behaviors

7/12/2018 • 2 minutes to read • [Edit Online](#)

Behaviors let you add functionality to user interface controls without having to subclass them. Instead, the functionality is implemented in a behavior class and attached to the control as if it was part of the control itself. This article provides an introduction to behaviors.

Behaviors enable you to implement code that you would normally have to write as code-behind, because it directly interacts with the API of the control in such a way that it can be concisely attached to the control and packaged for reuse across more than one application. They can be used to provide a full range of functionality to controls, such as:

- Adding an email validator to an [Entry](#).
- Creating a rating control using a tap gesture recognizer.
- Controlling an animation.
- Adding an effect to a control.

Behaviors also enable more advanced scenarios. In the context of *commanding*, behaviors are a useful approach for connecting a control to a command. In addition, they can be used to associate commands with controls that were not designed to interact with commands. For example, they can be used to invoke a command in response to an event firing.

Xamarin.Forms supports two different styles of behaviors:

- **Xamarin.Forms behaviors** – classes that derive from the [Behavior](#) or [Behavior<T>](#) class, where `T` is the type of the control to which the behavior should apply. For more information about Xamarin.Forms behaviors, see [Xamarin.Forms Behaviors](#) and [Reusable Behaviors](#).
- **Attached behaviors** – [static](#) classes with one or more attached properties. For more information about attached behaviors, see [Attached Behaviors](#).

This guide focuses on Xamarin.Forms behaviors because they are the preferred approach to behavior construction.

## Related Links

- [Behavior](#)
- [Behavior<T>](#)

# Attached Behaviors

7/12/2018 • 3 minutes to read • [Edit Online](#)

*Attached behaviors are static classes with one or more attached properties. This article demonstrates how to create and consume attached behaviors.*

## Overview

An attached property is a special type of bindable property. They are defined in one class but attached to other objects, and they are recognizable in XAML as attributes that contain a class and a property name separated by a period.

An attached property can define a `PropertyChanged` delegate that will be executed when the value of the property changes, such as when the property is set on a control. When the `PropertyChanged` delegate executes, it's passed a reference to the control on which it is being attached, and parameters that contain the old and new values for the property. This delegate can be used to add new functionality to the control that the property is attached to by manipulating the reference that is passed in, as follows:

1. The `PropertyChanged` delegate casts the control reference, which is received as a `BindableObject`, to the control type that the behavior is designed to enhance.
2. The `PropertyChanged` delegate modifies properties of the control, calls methods of the control, or registers event handlers for events exposed by the control, to implement the core behavior functionality.

An issue with attached behaviors is that they are defined in a `static` class, with `static` properties and methods. This makes it difficult to create attached behaviors that have state. In addition, Xamarin.Forms behaviors have replaced attached behaviors as the preferred approach to behavior construction. For more information about Xamarin.Forms behaviors, see [Xamarin.Forms Behaviors](#) and [Reusable Behaviors](#).

## Creating an Attached Behavior

The sample application demonstrates a `NumericValidationBehavior`, which highlights the value entered by the user into an `Entry` control in red, if it's not a `double`. The behavior is shown in the following code example:

```

public static class NumericValidationBehavior
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached (
            "AttachBehavior",
            typeof(bool),
            typeof(NumericValidationBehavior),
            false,
            propertyChanged:OnAttachBehaviorChanged);

    public static bool GetAttachBehavior (BindableObject view)
    {
        return (bool)view.GetValue (AttachBehaviorProperty);
    }

    public static void SetAttachBehavior (BindableObject view, bool value)
    {
        view.SetValue (AttachBehaviorProperty, value);
    }

    static void OnAttachBehaviorChanged (BindableObject view, object oldValue, object newValue)
    {
        var entry = view as Entry;
        if (entry == null) {
            return;
        }

        bool attachBehavior = (bool)newValue;
        if (attachBehavior) {
            entry.TextChanged += OnEntryTextChanged;
        } else {
            entry.TextChanged -= OnEntryTextChanged;
        }
    }

    static void OnEntryTextChanged (object sender, TextChangedEventArgs args)
    {
        double result;
        bool isValid = double.TryParse (args.NewTextValue, out result);
        ((Entry)sender).TextColor = isValid ? Color.Default : Color.Red;
    }
}

```

The `NumericValidationBehavior` class contains an attached property named `AttachBehavior` with a `static` getter and setter, which controls the addition or removal of the behavior to the control to which it will be attached. This attached property registers the `OnAttachBehaviorChanged` method that will be executed when the value of the property changes. This method registers or de-registers an event handler for the `TextChanged` event, based on the value of the `AttachBehavior` attached property. The core functionality of the behavior is provided by the `OnEntryTextChanged` method, which parses the value entered into the `Entry` by the user, and sets the `TextColor` property to red if the value isn't a `double`.

## Consuming an Attached Behavior

The `NumericValidationBehavior` class can be consumed by adding the `AttachBehavior` attached property to an `Entry` control, as demonstrated in the following XAML code example:

```

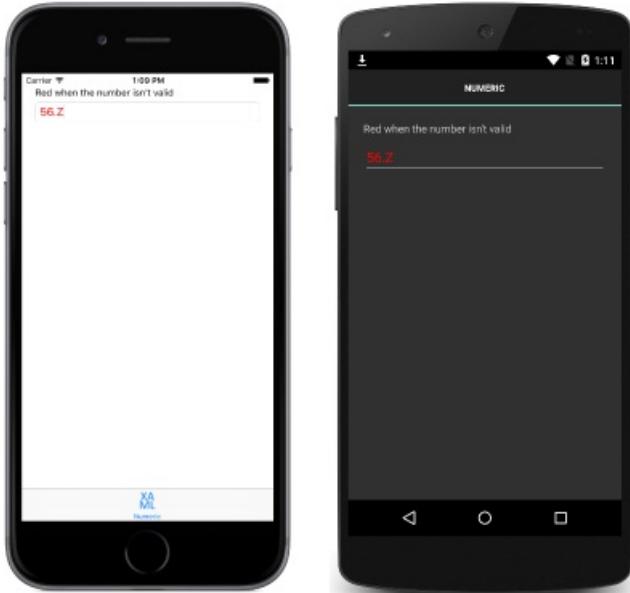
<ContentPage ... xmlns:local="clr-namespace:WorkingWithBehaviors;assembly=WorkingWithBehaviors" ...>
    ...
    <Entry Placeholder="Enter a System.Double" local:NumericValidationBehavior.AttachBehavior="true" />
    ...
</ContentPage>

```

The equivalent `Entry` in C# is shown in the following code example:

```
var entry = new Entry { Placeholder = "Enter a System.Double" };
NumericValidationBehavior.SetAttachBehavior (entry, true);
```

At runtime, the behavior will respond to interaction with the control, according to the behavior implementation. The following screenshots demonstrate the attached behavior responding to invalid input:



#### NOTE

Attached behaviors are written for a specific control type (or a superclass that can apply to many controls), and they should only be added to a compatible control. Attempting to attach a behavior to an incompatible control will result in unknown behavior, and depends on the behavior implementation.

## Removing an Attached Behavior from a Control

The `NumericValidationBehavior` class can be removed from a control by setting the `AttachBehavior` attached property to `false`, as demonstrated in the following XAML code example:

```
<Entry Placeholder="Enter a System.Double" local:NumericValidationBehavior.AttachBehavior="false" />
```

The equivalent `Entry` in C# is shown in the following code example:

```
var entry = new Entry { Placeholder = "Enter a System.Double" };
NumericValidationBehavior.SetAttachBehavior (entry, false);
```

At runtime, the `OnAttachBehaviorChanged` method will be executed when the value of the `AttachBehavior` attached property is set to `false`. The `OnAttachBehaviorChanged` method will then de-register the event handler for the `TextChanged` event, ensuring that the behavior isn't executed as the user interacts with the control.

## Summary

This article demonstrated how to create and consume attached behaviors. Attached behaviors are `static` classes with one or more attached properties.

## Related Links

- [Attached Behaviors \(sample\)](#)

# Xamarin.Forms Behaviors

7/12/2018 • 5 minutes to read • [Edit Online](#)

*Xamarin.Forms behaviors are created by deriving from the Behavior or Behavior class. This article demonstrates how to create and consume Xamarin.Forms behaviors.*

## Overview

The process for creating a Xamarin.Forms behavior is as follows:

1. Create a class that inherits from the `Behavior` or `Behavior<T>` class, where `T` is the type of the control to which the behavior should apply.
2. Override the `OnAttachedTo` method to perform any required setup.
3. Override the `OnDetachingFrom` method to perform any required cleanup.
4. Implement the core functionality of the behavior.

This results in the structure shown in the following code example:

```
public class CustomBehavior : Behavior<View>
{
    protected override void OnAttachedTo (View bindable)
    {
        base.OnAttachedTo (bindable);
        // Perform setup
    }

    protected override void OnDetachingFrom (View bindable)
    {
        base.OnDetachingFrom (bindable);
        // Perform clean up
    }

    // Behavior implementation
}
```

The `OnAttachedTo` method is fired immediately after the behavior is attached to a control. This method receives a reference to the control to which it is attached, and can be used to register event handlers or perform other setup that's required to support the behavior functionality. For example, you could subscribe to an event on a control. The behavior functionality would then be implemented in the event handler for the event.

The `OnDetachingFrom` method is fired when the behavior is removed from the control. This method receives a reference to the control to which it is attached, and is used to perform any required cleanup. For example, you could unsubscribe from an event on a control to prevent memory leaks.

The behavior can then be consumed by attaching it to the `Behaviors` collection of the appropriate control.

## Creating a Xamarin.Forms Behavior

The sample application demonstrates a `NumericValidationBehavior`, which highlights the value entered by the user into an `Entry` control in red, if it's not a `double`. The behavior is shown in the following code example:

```

public class NumericValidationBehavior : Behavior<Entry>
{
    protected override void OnAttachedTo(Entry entry)
    {
        entry.TextChanged += OnEntryTextChanged;
        base.OnAttachedTo(entry);
    }

    protected override void OnDetachingFrom(Entry entry)
    {
        entry.TextChanged -= OnEntryTextChanged;
        base.OnDetachingFrom(entry);
    }

    void OnEntryTextChanged(object sender, TextChangedEventArgs args)
    {
        double result;
        bool isValid = double.TryParse (args.NewTextValue, out result);
        ((Entry)sender).TextColor = isValid ? Color.Default : Color.Red;
    }
}

```

The `NumericValidationBehavior` derives from the `Behavior<T>` class, where `T` is an `Entry`. The `OnAttachedTo` method registers an event handler for the `TextChanged` event, with the `OnDetachingFrom` method de-registering the `TextChanged` event to prevent memory leaks. The core functionality of the behavior is provided by the `OnEntryTextChanged` method, which parses the value entered by the user into the `Entry`, and sets the `TextColor` property to red if the value isn't a `double`.

#### NOTE

Xamarin.Forms does not set the `BindingContext` of a behavior, because behaviors can be shared and applied to multiple controls through styles.

## Consuming a Xamarin.Forms Behavior

Every Xamarin.Forms control has a `Behaviors` collection, to which one or more behaviors can be added, as demonstrated in the following XAML code example:

```

<Entry Placeholder="Enter a System.Double">
    <Entry.Behaviors>
        <local:NumericValidationBehavior />
    </Entry.Behaviors>
</Entry>

```

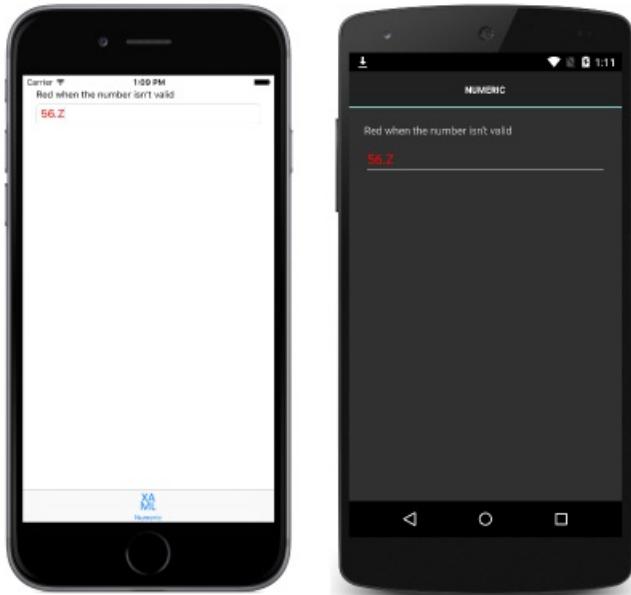
The equivalent `Entry` in C# is shown in the following code example:

```

var entry = new Entry { Placeholder = "Enter a System.Double" };
entry.Behaviors.Add (new NumericValidationBehavior ());

```

At runtime the behavior will respond to interaction with the control, according to the behavior implementation. The following screenshots demonstrate the behavior responding to invalid input:



#### NOTE

Behaviors are written for a specific control type (or a superclass that can apply to many controls), and they should only be added to a compatible control. Attempting to attach a behavior to an incompatible control will result in an exception being thrown.

### Consuming a Xamarin.Forms Behavior with a Style

Behaviors can also be consumed by an explicit or implicit style. However, creating a style that sets the `Behaviors` property of a control is not possible because the property is read-only. The solution is to add an attached property to the behavior class that controls adding and removing the behavior. The process is as follows:

1. Add an attached property to the behavior class that will be used to control the addition or removal of the behavior to the control to which the behavior will attach. Ensure that the attached property registers a `PropertyChanged` delegate that will be executed when the value of the property changes.
2. Create a `static` getter and setter for the attached property.
3. Implement logic in the `PropertyChanged` delegate to add and remove the behavior.

The following code example shows an attached property that controls adding and removing the

`NumericValidationBehavior`:

```

public class NumericValidationBehavior : Behavior<Entry>
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached ("AttachBehavior", typeof(bool), typeof(NumericValidationBehavior),
        false, propertyChanged: OnAttachBehaviorChanged);

    public static bool GetAttachBehavior (BindableObject view)
    {
        return (bool)view.GetValue (AttachBehaviorProperty);
    }

    public static void SetAttachBehavior (BindableObject view, bool value)
    {
        view.SetValue (AttachBehaviorProperty, value);
    }

    static void OnAttachBehaviorChanged (BindableObject view, object oldValue, object newValue)
    {
        var entry = view as Entry;
        if (entry == null) {
            return;
        }

        bool attachBehavior = (bool)newValue;
        if (attachBehavior) {
            entry.Behaviors.Add (new NumericValidationBehavior ());
        } else {
            var toRemove = entry.Behaviors.FirstOrDefault (b => b is NumericValidationBehavior);
            if (toRemove != null) {
                entry.Behaviors.Remove (toRemove);
            }
        }
    }
    ...
}

```

The `NumericValidationBehavior` class contains an attached property named `AttachBehavior` with a `static` getter and setter, which controls the addition or removal of the behavior to the control to which it will be attached. This attached property registers the `OnAttachBehaviorChanged` method that will be executed when the value of the `AttachBehavior` attached property changes. This method adds or removes the behavior to the control, based on the value of the `AttachBehavior` attached property.

The following code example shows an *explicit* style for the `NumericValidationBehavior` that uses the `AttachBehavior` attached property, and which can be applied to `Entry` controls:

```

<Style x:Key="NumericValidationStyle" TargetType="Entry">
    <Style.Setters>
        <Setter Property="local:NumericValidationBehavior.AttachBehavior" Value="true" />
    </Style.Setters>
</Style>

```

The `Style` can be applied to an `Entry` control by setting its `Style` property to the `Style` instance using the `StaticResource` markup extension, as demonstrated in the following code example:

```

<Entry Placeholder="Enter a System.Double" Style="{StaticResource NumericValidationStyle}">

```

For more information about styles, see [Styles](#).

#### NOTE

While you can add bindable properties to a behavior that is set or queried in XAML, if you do create behaviors that have state they should not be shared between controls in a `Style` in a `ResourceDictionary`.

### Removing a Behavior from a Control

The `OnDetachingFrom` method is fired when a behavior is removed from a control, and is used to perform any required cleanup such as unsubscribing from an event to prevent a memory leak. However, behaviors are not implicitly removed from controls unless the control's `Behaviors` collection is modified by a `Remove` or `Clear` method. The following code example demonstrates removing a specific behavior from a control's `Behaviors` collection:

```
var toRemove = entry.Behaviors.FirstOrDefault (b => b is NumericValidationBehavior);
if (toRemove != null) {
    entry.Behaviors.Remove (toRemove);
}
```

Alternatively, the control's `Behaviors` collection can be cleared, as demonstrated in the following code example:

```
entry.Behaviors.Clear();
```

In addition, note that behaviors are not implicitly removed from controls when pages are popped from the navigation stack. Instead, they must be explicitly removed prior to pages going out of scope.

## Summary

This article demonstrated how to create and consume Xamarin.Forms behaviors. Xamarin.Forms behaviors are created by deriving from the `Behavior` or `Behavior<T>` class.

## Related Links

- [Xamarin.Forms Behavior \(sample\)](#)
- [Xamarin.Forms Behavior applied with a Style \(sample\)](#)
- [Behavior](#)
- [Behavior](#)

# Reusable Behaviors

11/1/2018 • 2 minutes to read • [Edit Online](#)

*Behaviors are reusable across more than one application. These articles explain how to create useful behaviors to perform commonly used functionality.*

## Reusable EffectBehavior

Behaviors are a useful approach for adding an effect to a control, removing boiler-plate effect handling code from code-behind files. This article demonstrates creating and consuming a Xamarin.Forms behavior to add an effect to a control.

## Reusable EventToCommandBehavior

Behaviors can be used to associate commands with controls that were not designed to interact with commands. This article demonstrates creating and consuming a Xamarin.Forms behavior to invoke a command when an event fires.

# Reusable EffectBehavior

11/1/2018 • 4 minutes to read • [Edit Online](#)

Behaviors are a useful approach for adding an effect to a control, removing boiler-plate effect handling code from code-behind files. This article demonstrates creating and consuming a Xamarin.Forms behavior to add an effect to a control.

## Overview

The `EffectBehavior` class is a reusable Xamarin.Forms custom behavior that adds an `Effect` instance to a control when the behavior is attached to the control, and removes the `Effect` instance when the behavior is detached from the control.

The following behavior properties must be set to use the behavior:

- **Group** – the value of the `ResolutionGroupName` attribute for the effect class.
- **Name** – the value of the `ExportEffect` attribute for the effect class.

For more information about effects, see [Effects](#).

### NOTE

The `EffectBehavior` is a custom class that can be located in the [Effect Behavior sample](#), and is not part of Xamarin.Forms.

## Creating the Behavior

The `EffectBehavior` class derives from the `Behavior<T>` class, where `T` is a `View`. This means that the `EffectBehavior` class can be attached to any Xamarin.Forms control.

### Implementing Bindable Properties

The `EffectBehavior` class defines two `BindableProperty` instances, which are used to add an `Effect` to a control when the behavior is attached to the control. These properties are shown in the following code example:

```
public class EffectBehavior : Behavior<View>
{
    public static readonly BindableProperty GroupProperty =
        BindableProperty.Create ("Group", typeof(string), typeof(EffectBehavior), null);
    public static readonly BindableProperty NameProperty =
        BindableProperty.Create ("Name", typeof(string), typeof(EffectBehavior), null);

    public string Group {
        get { return (string)GetValue (GroupProperty); }
        set { SetValue (GroupProperty, value); }
    }

    public string Name {
        get { return (string)GetValue (NameProperty); }
        set { SetValue (NameProperty, value); }
    }
    ...
}
```

When the `EffectBehavior` is consumed, the `Group` property should be set to the value of the `ResolutionGroupName`

attribute for the effect. In addition, the `Name` property should be set to the value of the `ExportEffect` attribute for the effect class.

## Implementing the Overrides

The `EffectBehavior` class overrides the `OnAttachedTo` and `OnDetachingFrom` methods of the `Behavior<T>` class, as shown in the following code example:

```
public class EffectBehavior : Behavior<View>
{
    ...
    protected override void OnAttachedTo (BindableObject bindable)
    {
        base.OnAttachedTo (bindable);
        AddEffect (bindable as View);
    }

    protected override void OnDetachingFrom (BindableObject bindable)
    {
        RemoveEffect (bindable as View);
        base.OnDetachingFrom (bindable);
    }
    ...
}
```

The `OnAttachedTo` method performs setup by calling the `AddEffect` method, passing in the attached control as a parameter. The `OnDetachingFrom` method performs cleanup by calling the `RemoveEffect` method, passing in the attached control as a parameter.

## Implementing the Behavior Functionality

The purpose of the behavior is to add the `Effect` defined in the `Group` and `Name` properties to a control when the behavior is attached to the control, and remove the `Effect` when the behavior is detached from the control. The core behavior functionality is shown in the following code example:

```
public class EffectBehavior : Behavior<View>
{
    ...
    void AddEffect (View view)
    {
        var effect = GetEffect ();
        if (effect != null) {
            view.Effects.Add (GetEffect ());
        }
    }

    void RemoveEffect (View view)
    {
        var effect = GetEffect ();
        if (effect != null) {
            view.Effects.Remove (GetEffect ());
        }
    }

    Effect GetEffect ()
    {
        if (!string.IsNullOrWhiteSpace (Group) && !string.IsNullOrWhiteSpace (Name)) {
            return Effect.Resolve (string.Format ("{0}.{1}", Group, Name));
        }
        return null;
    }
}
```

The `AddEffect` method is executed in response to the `EffectBehavior` being attached to a control, and it receives the attached control as a parameter. The method then adds the retrieved effect to the control's `Effects` collection. The `RemoveEffect` method is executed in response to the `EffectBehavior` being detached from a control, and it receives the attached control as a parameter. The method then removes the effect from the control's `Effects` collection.

The `GetEffect` method uses the `Effect.Resolve` method to retrieve the `Effect`. The effect is located through a concatenation of the `Group` and `Name` property values. If a platform doesn't provide the effect, the `Effect.Resolve` method will return a non-`null` value.

## Consuming the Behavior

The `EffectBehavior` class can be attached to the `Behaviors` collection of a control, as demonstrated in the following XAML code example:

```
<Label Text="Label Shadow Effect" ...>
  <Label.Behaviors>
    <local:EffectBehavior Group="Xamarin" Name="LabelShadowEffect" />
  </Label.Behaviors>
</Label>
```

The equivalent C# code is shown in the following code example:

```
var label = new Label {
  Text = "Label Shadow Effect",
  ...
};
label.Behaviors.Add (new EffectBehavior {
  Group = "Xamarin",
  Name = "LabelShadowEffect"
});
```

The `Group` and `Name` properties of the behavior are set to the values of the `ResolutionGroupName` and `ExportEffect` attributes for the effect class in each platform-specific project.

At runtime, when the behavior is attached to the `Label` control, the `Xamarin.LabelShadowEffect` will be added to the control's `Effects` collection. This results in a shadow being added to the text displayed by the `Label` control, as shown in the following screenshots:

Label Shadow Effect

iOS

Label Shadow Effect

Android

The advantage of using this behavior to add and remove effects from controls is that boiler-plate effect-handling code can be removed from code-behind files.

## Summary

This article demonstrated using a behavior to add an effect to a control. The `EffectBehavior` class is a reusable Xamarin.Forms custom behavior that adds an `Effect` instance to a control when the behavior is attached to the control, and removes the `Effect` instance when the behavior is detached from the control.

## Related Links

- [Effects](#)
- [Effect Behavior \(sample\)](#)
- [Behavior](#)
- [Behavior](#)

# Reusable EventToCommandBehavior

11/12/2018 • 6 minutes to read • [Edit Online](#)

Behaviors can be used to associate commands with controls that were not designed to interact with commands. This article demonstrates creating and consuming a Xamarin.Forms behavior to invoke a command when an event fires.

## Overview

The `EventToCommandBehavior` class is a reusable Xamarin.Forms custom behavior that executes a command in response to *any* event firing. By default, the event arguments for the event will be passed to the command, and can be optionally converted by an `IValueConverter` implementation.

The following behavior properties must be set to use the behavior:

- **EventName** – the name of the event the behavior listens to.
- **Command** – the `ICommand` to be executed. The behavior expects to find the `ICommand` instance on the `BindingContext` of the attached control, which may be inherited from a parent element.

The following optional behavior properties can also be set:

- **CommandParameter** – an `object` that will be passed to the command.
- **Converter** – an `IValueConverter` implementation that will change the format of the event argument data as it's passed between *source* and *target* by the binding engine.

### NOTE

The `EventToCommandBehavior` is a custom class that can be located in the [EventToCommand Behavior sample](#), and is not part of Xamarin.Forms.

## Creating the Behavior

The `EventToCommandBehavior` class derives from the `BehaviorBase<T>` class, which in turn derives from the `Behavior<T>` class. The purpose of the `BehaviorBase<T>` class is to provide a base class for any Xamarin.Forms behaviors that require the `BindingContext` of the behavior to be set to the attached control. This ensures that the behavior can bind to and execute the `ICommand` specified by the `Command` property when the behavior is consumed.

The `BehaviorBase<T>` class provides an overridable `OnAttachedTo` method that sets the `BindingContext` of the behavior and an overridable `OnDetachingFrom` method that cleans up the `BindingContext`. In addition, the class stores a reference to the attached control in the `AssociatedObject` property.

### Implementing Bindable Properties

The `EventToCommandBehavior` class defines four `BindableProperty` instances, that execute a user defined command when an event fires. These properties are shown in the following code example:

```

public class EventToCommandBehavior : BehaviorBase<View>
{
    public static readonly BindableProperty EventNameProperty =
        BindableProperty.Create ("EventName", typeof(string), typeof(EventToCommandBehavior), null,
        propertyChanged: OnEventNameChanged);
    public static readonly BindableProperty CommandProperty =
        BindableProperty.Create ("Command", typeof(ICommand), typeof(EventToCommandBehavior), null);
    public static readonly BindableProperty CommandParameterProperty =
        BindableProperty.Create ("CommandParameter", typeof(object), typeof(EventToCommandBehavior), null);
    public static readonly BindableProperty InputConverterProperty =
        BindableProperty.Create ("Converter", typeof(IValueConverter), typeof(EventToCommandBehavior), null);

    public string EventName { ... }
    public ICommand Command { ... }
    public object CommandParameter { ... }
    public IValueConverter Converter { ... }
    ...
}

```

When the `EventToCommandBehavior` class is consumed, the `Command` property should be data bound to an `ICommand` to be executed in response to the event firing that's defined in the `EventName` property. The behavior will expect to find the `ICommand` on the `BindingContext` of the attached control.

By default, the event arguments for the event will be passed to the command. This data can be optionally converted as it's passed between *source* and *target* by the binding engine, by specifying an `IValueConverter` implementation as the `Converter` property value. Alternatively, a parameter can be passed to the command by specifying the `CommandParameter` property value.

## Implementing the Overrides

The `EventToCommandBehavior` class overrides the `OnAttachedTo` and `OnDetachingFrom` methods of the `BehaviorBase<T>` class, as shown in the following code example:

```

public class EventToCommandBehavior : BehaviorBase<View>
{
    ...
    protected override void OnAttachedTo (View bindable)
    {
        base.OnAttachedTo (bindable);
        RegisterEvent (EventName);
    }

    protected override void OnDetachingFrom (View bindable)
    {
        DeregisterEvent (EventName);
        base.OnDetachingFrom (bindable);
    }
    ...
}

```

The `OnAttachedTo` method performs setup by calling the `RegisterEvent` method, passing in the value of the `EventName` property as a parameter. The `OnDetachingFrom` method performs cleanup by calling the `DeregisterEvent` method, passing in the value of the `EventName` property as a parameter.

## Implementing the Behavior Functionality

The purpose of the behavior is to execute the command defined by the `Command` property in response to the event firing that's defined by the `EventName` property. The core behavior functionality is shown in the following code example:

```

public class EventToCommandBehavior : BehaviorBase<View>
{
    ...
    void RegisterEvent (string name)
    {
        if (string.IsNullOrWhiteSpace (name)) {
            return;
        }

        EventInfo eventInfo = AssociatedObject.GetType ().GetRuntimeEvent (name);
        if (eventInfo == null) {
            throw new ArgumentException (string.Format ("EventToCommandBehavior: Can't register the '{0}' event.", EventName));
        }
        MethodInfo methodInfo = typeof(EventToCommandBehavior).GetTypeInfo ().GetDeclaredMethod ("OnEvent");
        eventHandler = methodInfo.CreateDelegate (eventInfo.EventHandlerType, this);
        eventInfo.AddEventHandler (AssociatedObject, eventHandler);
    }

    void OnEvent (object sender, object eventArgs)
    {
        if (Command == null) {
            return;
        }

        object resolvedParameter;
        if (CommandParameter != null) {
            resolvedParameter = CommandParameter;
        } else if (Converter != null) {
            resolvedParameter = Converter.Convert (eventArgs, typeof(object), null, null);
        } else {
            resolvedParameter = eventArgs;
        }

        if (Command.CanExecute (resolvedParameter)) {
            Command.Execute (resolvedParameter);
        }
    }
    ...
}

```

The `RegisterEvent` method is executed in response to the `EventToCommandBehavior` being attached to a control, and it receives the value of the `EventName` property as a parameter. The method then attempts to locate the event defined in the `EventName` property, on the attached control. Provided that the event can be located, the `OnEvent` method is registered to be the handler method for the event.

The `OnEvent` method is executed in response to the event firing that's defined in the `EventName` property. Provided that the `Command` property references a valid `ICommand`, the method attempts to retrieve a parameter to pass to the `ICommand` as follows:

- If the `CommandParameter` property defines a parameter, it is retrieved.
- Otherwise, if the `Converter` property defines an `IValueConverter` implementation, the converter is executed and converts the event argument data as it's passed between *source* and *target* by the binding engine.
- Otherwise, the event arguments are assumed to be the parameter.

The data bound `ICommand` is then executed, passing in the parameter to the command, provided that the `CanExecute` method returns `true`.

Although not shown here, the `EventToCommandBehavior` also includes a `DeregisterEvent` method that's executed by the `OnDetachingFrom` method. The `DeregisterEvent` method is used to locate and deregister the event defined in the `EventName` property, to cleanup any potential memory leaks.

## Consuming the Behavior

The `EventToCommandBehavior` class can be attached to the `Behaviors` collection of a control, as demonstrated in the following XAML code example:

```
<ListView ItemsSource="{Binding People}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextCell Text="{Binding Name}" />
        </DataTemplate>
    </ListView.ItemTemplate>
    <ListView.Behaviors>
        <local:EventToCommandBehavior EventName="ItemSelected" Command="{Binding OutputAgeCommand}"
        Converter="{StaticResource SelectedItemConverter}" />
    </ListView.Behaviors>
</ListView>
<Label Text="{Binding SelectedItemText}" />
```

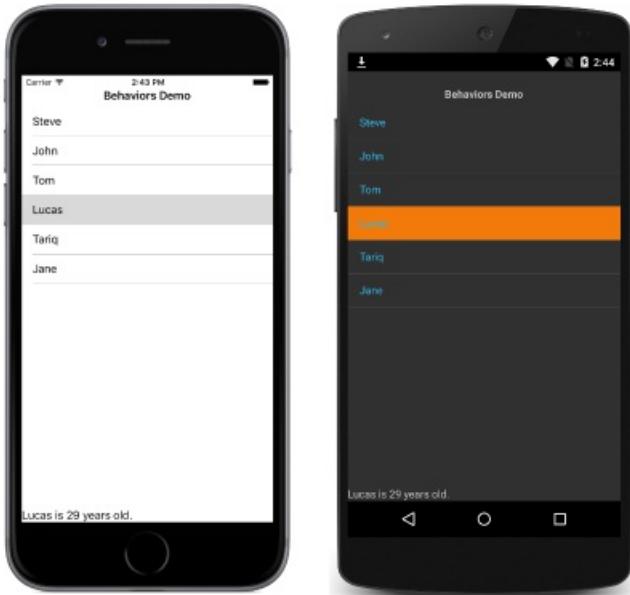
The equivalent C# code is shown in the following code example:

```
var listView = new ListView();
listView.SetBinding(ItemsView<Cell>.ItemsSourceProperty, "People");
listView.ItemTemplate = new DataTemplate(() =>
{
    var textCell = new TextCell();
    textCell.SetBinding(TextCell.TextProperty, "Name");
    return textCell;
});
listView.Behaviors.Add(new EventToCommandBehavior
{
    EventName = "ItemSelected",
    Command = ((HomePageViewModel)BindingContext).OutputAgeCommand,
    Converter = new SelectedItemEventArgsToSelectedItemConverter()
});

var selectedItemLabel = new Label();
selectedItemLabel.SetBinding(Label.TextProperty, "SelectedItemText");
```

The `Command` property of the behavior is data bound to the `OutputAgeCommand` property of the associated ViewModel, while the `Converter` property is set to the `SelectedItemConverter` instance, which returns the `SelectedItem` of the `ListView` from the `SelectedItemChangedEventArgs`.

At runtime, the behavior will respond to interaction with the control. When an item is selected in the `ListView`, the `ItemSelected` event will fire, which will execute the `OutputAgeCommand` in the ViewModel. In turn this updates the ViewModel `SelectedItemText` property that the `Label` binds to, as shown in the following screenshots:



The advantage of using this behavior to execute a command when an event fires, is that commands can be associated with controls that weren't designed to interact with commands. In addition, this removes boiler-plate event handling code from code-behind files.

## Summary

This article demonstrated using a Xamarin.Forms behavior to invoke a command when an event fires. Behaviors can be used to associate commands with controls that were not designed to interact with commands.

## Related Links

- [EventToCommand Behavior \(sample\)](#)
- [Behavior](#)
- [Behavior<T>](#)

# Xamarin.Forms Custom Renderers

7/12/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms user interfaces are rendered using the native controls of the target platform, allowing Xamarin.Forms applications to retain the appropriate look and feel for each platform. Custom Renderers let developers override this process to customize the appearance and behavior of Xamarin.Forms controls on each platform.*

## Introduction to custom renderers

Custom renderers provide a powerful approach for customizing the appearance and behavior of Xamarin.Forms controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization. This article provides an introduction to custom renderers, and outlines the process for creating a custom renderer.

## Renderer base classes and native controls

Every Xamarin.Forms control has an accompanying renderer for each platform that creates an instance of a native control. This article lists the renderer and native control classes that implement each Xamarin.Forms page, layout, view, and cell.

## Customizing an Entry

The Xamarin.Forms `Entry` control allows a single line of text to be edited. This article demonstrates how to create a custom renderer for the `Entry` control, enabling developers to override the default native rendering with their own platform-specific customization.

## Customizing a ContentPage

A `ContentPage` is a visual element that displays a single view and occupies most of the screen. This article demonstrates how to create a custom renderer for the `ContentPage` page, enabling developers to override the default native rendering with their own platform-specific customization.

## Customizing a Map

Xamarin.Forms.Maps provides a cross-platform abstraction for displaying maps that use the native map APIs on each platform, to provide a fast and familiar map experience for users. This topic demonstrates how to create custom renderers for the `Map` control, enabling developers to override the default native rendering with their own platform-specific customization.

## Customizing a ListView

A Xamarin.Forms `ListView` is a view that displays a collection of data as a vertical list. This article demonstrates how to create a custom renderer that encapsulates platform-specific list controls and native cell layouts, allowing more control over native list control performance.

## Customizing a ViewCell

A Xamarin.Forms `ViewCell` is a cell that can be added to a `ListView` or `TableView`, which contains a developer-

defined view. This article demonstrates how to create a custom renderer for a `ViewCell` that's hosted inside a Xamarin.Forms `ListView` control. This stops the Xamarin.Forms layout calculations from being repeatedly called during `ListView` scrolling.

## Implementing a View

Xamarin.Forms custom user interfaces controls should derive from the `View` class, which is used to place layouts and controls on the screen. This article demonstrates how to create a custom renderer for a Xamarin.Forms custom control that's used to display a preview video stream from the device's camera.

## Implementing a HybridWebView

This article demonstrates how to create a custom renderer for a `HybridWebView` custom control, which demonstrates how to enhance the platform-specific web controls to allow C# code to be invoked from JavaScript.

## Implementing a video player

This article shows how to write renderers to implement a custom `VideoPlayer` control that can play videos from the web, videos embedded as application resources, or videos stored in the video library on the user's device. Several techniques are demonstrated, including implementing methods and read-only bindable properties.

## Related Links

- [Effects](#)
- [Custom Renderers \(Xamarin University Video\)](#)
- [Custom Renderers \(Xamarin University Video\) Sample](#)

# Introduction to Custom Renderers

10/9/2018 • 4 minutes to read • [Edit Online](#)

*Custom renderers provide a powerful approach for customizing the appearance and behavior of Xamarin.Forms controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization. This article provides an introduction to custom renderers, and outlines the process for creating a custom renderer.*

Xamarin.Forms [Pages, Layouts and Controls](#) present a common API to describe cross-platform mobile user interfaces. Each page, layout, and control is rendered differently on each platform, using a `Renderer` class that in turn creates a native control (corresponding to the Xamarin.Forms representation), arranges it on the screen, and adds the behavior specified in the shared code.

Developers can implement their own custom `Renderer` classes to customize the appearance and/or behavior of a control. Custom renderers for a given type can be added to one application project to customize the control in one place while allowing the default behavior on other platforms; or different custom renderers can be added to each application project to create a different look and feel on iOS, Android, and the Universal Windows Platform (UWP). However, implementing a custom renderer class to perform a simple control customization is often a heavy-weight response. Effects simplify this process, and are typically used for small styling changes. For more information, see [Effects](#).

## Examining Why Custom Renderers are Necessary

Changing the appearance of a Xamarin.Forms control, without using a custom renderer, is a two-step process that involves creating a custom control through subclassing, and then consuming the custom control in place of the original control. The following code example shows an example of subclassing the `Entry` control:

```
public class MyEntry : Entry
{
    public MyEntry ()
    {
        BackgroundColor = Color.Gray;
    }
}
```

The `MyEntry` control is an `Entry` control where the `BackgroundColor` is set to gray, and can be referenced in Xaml by declaring a namespace for its location and using the namespace prefix on the control element. The following code example shows how the `MyEntry` custom control can be consumed by a `ContentPage`:

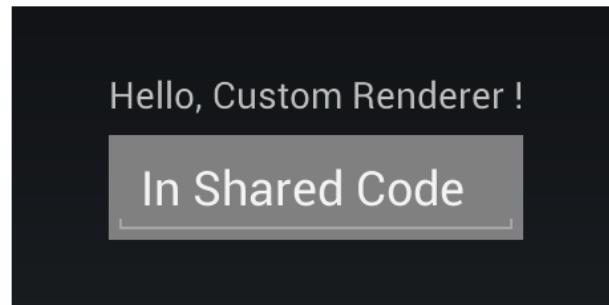
```
<ContentPage
    ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```

The `local` namespace prefix can be anything. However, the `namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom control.

#### NOTE

Defining the `xm1ns` is much simpler in .NET Standard library projects than Shared Projects. A .NET Standard library is compiled into an assembly so it's easy to determine what the `assembly=CustomRenderer` value should be. When using Shared Projects, all the shared assets (including the XAML) are compiled into each of the referencing projects, which means that if the iOS, Android, and UWP projects have their own *assembly names* it is impossible to write the `xm1ns` declaration because the value needs to be different for each application. Custom controls in XAML for Shared Projects will require every application project to be configured with the same assembly name.

The `MyEntry` custom control is then rendered on each platform, with a gray background, as shown in the following screenshots:



iOS

Android

Changing the background color of the control on each platform has been accomplished purely through subclassing the control. However, this technique is limited in what it can achieve as it is not possible to take advantage of platform-specific enhancements and customizations. When they are required, custom renderers must be implemented.

## Creating a Custom Renderer Class

The process for creating a custom renderer class is as follows:

1. Create a subclass of the renderer class that renders the native control.
2. Override the method that renders the native control and write logic to customize the control. Often, the `OnElementChanged` method is used to render the native control, which is called when the corresponding `Xamarin.Forms` control is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the `Xamarin.Forms` control. This attribute is used to register the custom renderer with `Xamarin.Forms`.

#### NOTE

For most `Xamarin.Forms` elements, it is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used. However, custom renderers are required in each platform project when rendering a `View` or `ViewCell` element.

The topics in this series will provide demonstrations and explanations of this process for different `Xamarin.Forms` elements.

## Troubleshooting

If a custom control is contained in a .NET Standard library project that's been added to the solution (i.e. not the .NET Standard library created by the Visual Studio for Mac/Visual Studio `Xamarin.Forms App` project template), an

exception may occur in iOS when attempting to access the custom control. If this issue occurs it can be resolved by creating a reference to the custom control from the `AppDelegate` class:

```
var temp = new ClassInPCL(); // in AppDelegate, but temp not used anywhere
```

This forces the compiler to recognize the `ClassInPCL` type by resolving it. Alternatively, the `Preserve` attribute can be added to the `AppDelegate` class to achieve the same result:

```
[assembly: Preserve (typeof (ClassInPCL))]
```

This creates a reference to the `ClassInPCL` type, indicating that it's required at runtime. For more information, see [Preserving Code](#).

## Summary

This article has provided an introduction to custom renderers, and has outlined the process for creating a custom renderer. Custom renderers provide a powerful approach for customizing the appearance and behavior of Xamarin.Forms controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization.

## Related Links

- [Effects](#)

# Renderer Base Classes and Native Controls

11/20/2018 • 2 minutes to read • [Edit Online](#)

*Every Xamarin.Forms control has an accompanying renderer for each platform that creates an instance of a native control. This article lists the renderer and native control classes that implement each Xamarin.Forms page, layout, view, and cell.*

With the exception of the `MapRenderer` class, the platform-specific renderers can be found in the following namespaces:

- **iOS** – `Xamarin.Forms.Platform.iOS`
- **Android** – `Xamarin.Forms.Platform.Android`
- **Android (AppCompat)** – `Xamarin.Forms.Platform.Android.AppCompat`
- **Universal Windows Platform (UWP)** – `Xamarin.Forms.Platform.UWP`

The `MapRenderer` class can be found in the following namespaces:

- **iOS** – `Xamarin.Forms.Maps.iOS`
- **Android** – `Xamarin.Forms.Maps.Android`
- **Universal Windows Platform (UWP)** – `Xamarin.Forms.Maps.UWP`

## Pages

The following table lists the renderer and native control classes that implement each Xamarin.Forms `Page` type:

PAGE	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
<a href="#">ContentPage</a>	<a href="#">PageRenderer</a>	<code>UIViewController</code>	<code>ViewGroup</code>		<code>FrameworkElement</code>
<a href="#">MasterDetailPage</a>	<code>PhoneMasterDetailRenderer (iOS – Phone), TabletMasterDetailPageRenderer (iOS – Tablet), MasterDetailRenderer (Android), MasterDetailPageRenderer (Android AppCompat), MasterDetailPageRenderer (UWP)</code>	<code>UIViewController (Phone), UISplitViewController (Tablet)</code>	<code>DrawerLayout (v4)</code>	<code>DrawerLayout (v4)</code>	<code>FrameworkElement (Custom Control)</code>
<a href="#">NavigationPage</a>	<code>NavigationRenderer (iOS and Android), NavigationPageRenderer (Android AppCompat), NavigationPageRenderer (UWP)</code>	<code>UIToolbar</code>	<code>ViewGroup</code>	<code>ViewGroup</code>	<code>FrameworkElement (Custom Control)</code>

PAGE	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
TabbedPage	TabbedRenderer (iOS and Android), TabPageRender er (Android AppCompat), TabPageRender er (UWP)	UIView	ViewPager	ViewPager	FrameworkEle ment (Pivot)
TemplatedPage	PageRenderer	UIViewController	ViewGroup		FrameworkEle ment
CarouselPage	CarouselPageRen derer	UIScrollView	ViewPager	ViewPager	FrameworkEle ment (FlipView)

## Layouts

The following table lists the renderer and native control classes that implement each Xamarin.Forms [Layout](#) type:

LAYOUT	RENDERER	IOS	ANDROID	UWP
ContentPresenter	ViewRenderer	UIView	View	FrameworkElement
ContentView	ViewRenderer	UIView	View	FrameworkElement
FlexLayout	ViewRenderer	UIView	View	FrameworkElement
Frame	FrameRenderer	UIView	ViewGroup	Border
ScrollView	ScrollViewRenderer	UIScrollView	ScrollView	ScrollViewer
TemplatedView	ViewRenderer	UIView	View	FrameworkElement
AbsoluteLayout	ViewRenderer	UIView	View	FrameworkElement
Grid	ViewRenderer	UIView	View	FrameworkElement
RelativeLayout	ViewRenderer	UIView	View	FrameworkElement
StackLayout	ViewRenderer	UIView	View	FrameworkElement

## Views

The following table lists the renderer and native control classes that implement each Xamarin.Forms [View](#) type:

VIEWS	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
ActivityIndicator	ActivityIndicatorR enderer	UIActivityIndicatorView	ProgressBar		ProgressBar

IEWS	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
<code>BoxView</code>	BoxRenderer (iOS and Android), BoxViewRenderer (UWP)	UIView	ViewGroup		Rectangle
<code>Button</code>	ButtonRenderer	UIButton	Button	AppCompatButton	Button
<code>DatePicker</code>	DatePickerRenderer	UITextField	EditText		DatePicker
<code>Editor</code>	EditorRenderer	UITextView	EditText		TextBox
<code>Entry</code>	EntryRenderer	UITextField	EditText		TextBox
<code>Image</code>	ImageRenderer	UIImageView	ImageView		Image
<code>ImageButton</code>	ImageButtonRenderer	UIButton		AppCompatImageButton	Button
<code>Label</code>	LabelRenderer	UILabel	TextView		TextBlock
<code>ListView</code>	ListViewRenderer	UITableView	ListView		ListView
<code>Map</code>	MapRenderer	MKMapView	MapView		MapControl
<code>Picker</code>	PickerRenderer	UITextField	EditText	EditText	ComboBox
<code>ProgressBar</code>	ProgressBarRenderer	UIProgressView	ProgressBar		ProgressBar
<code>earchBar</code>	SearchBarRenderer	UISearchBar	SearchView		AutoSuggestBox
<code>Slider</code>	SliderRenderer	UISlider	SeekBar		Slider
<code>Stepper</code>	StepperRenderer	UIStepper	LinearLayout		Control
<code>Switch</code>	SwitchRenderer	UISwitch	Switch	SwitchCompat	ToggleSwitch
<code>TableView</code>	TableViewRenderer	UITableView	ListView		ListView
<code>TimePicker</code>	TimePickerRenderer	UITextField	EditText		TimePicker
<code>WebView</code>	WebViewRenderer	UIWebView	WebView		WebView

## Cells

The following table lists the renderer and native control classes that implement each Xamarin.Forms [Cell](#) type:

CELLS	RENDERER	IOS	ANDROID	UWP
<a href="#">EntryCell</a>	EntryCellRenderer	UITableViewController with a UITextField	LinearLayout with a TextView and EditText	DataTemplate with a TextBox
<a href="#">SwitchCell</a>	SwitchCellRenderer	UITableViewController with a UISwitch	Switch	DataTemplate with a Grid containing a TextBlock and ToggleSwitch
<a href="#">TextCell</a>	TextCellRenderer	UITableViewController	LinearLayout with two TextViews	DataTemplate with a StackPanel containing two TextBlocks
<a href="#">ImageCell</a>	ImageCellRenderer	UITableViewController with a UIImage	LinearLayout with two TextViews and an ImageView	DataTemplate with a Grid containing an Image and two TextBlocks
<a href="#">ViewCell</a>	ViewCellRenderer	UITableViewController	View	DataTemplate with a ContentPresenter

## Summary

This article has listed the renderer and native control classes that implement each Xamarin.Forms page, layout, view, and cell. Every Xamarin.Forms control has an accompanying renderer for each platform that creates an instance of a native control.

## Related Links

- [Custom Renderers \(Xamarin University Video\)](#)

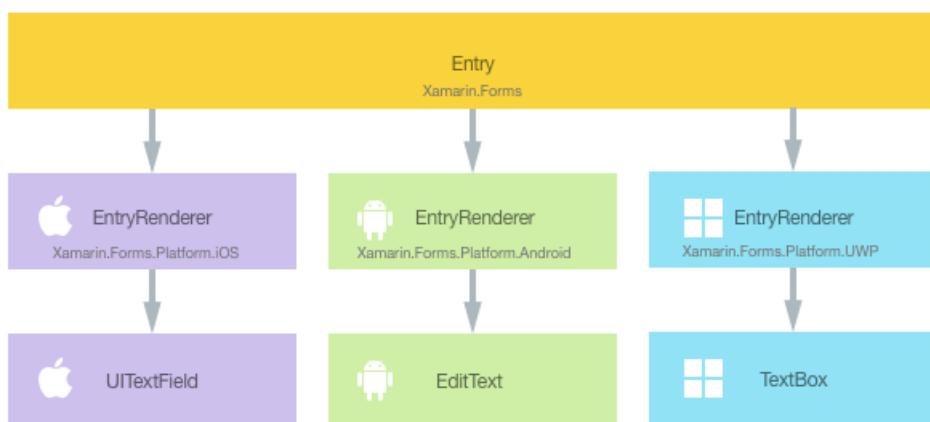
# Customizing an Entry

7/12/2018 • 6 minutes to read • [Edit Online](#)

The `Xamarin.Forms Entry` control allows a single line of text to be edited. This article demonstrates how to create a custom renderer for the `Entry` control, enabling developers to override the default native rendering with their own platform-specific customization.

Every `Xamarin.Forms` control has an accompanying renderer for each platform that creates an instance of a native control. When an `Entry` control is rendered by a `Xamarin.Forms` application, in iOS the `EntryRenderer` class is instantiated, which in turns instantiates a native `UITextField` control. On the Android platform, the `EntryRenderer` class instantiates an `EditText` control. On the Universal Windows Platform (UWP), the `EntryRenderer` class instantiates a `TextBox` control. For more information about the renderer and native control classes that `Xamarin.Forms` controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `Entry` control and the corresponding native controls that implement it:



The rendering process can be taken advantage of to implement platform-specific customizations by creating a custom renderer for the `Entry` control on each platform. The process for doing this is as follows:

1. [Create](#) a `Xamarin.Forms` custom control.
2. [Consume](#) the custom control from `Xamarin.Forms`.
3. [Create](#) the custom renderer for the control on each platform.

Each item will now be discussed in turn, to implement an `Entry` control that has a different background color on each platform.

## Creating the Custom Entry Control

A custom `Entry` control can be created by subclassing the `Entry` control, as shown in the following code example:

```
public class MyEntry : Entry
{}
```

The `MyEntry` control is created in the .NET Standard library project and is simply an `Entry` control. Customization

of the control will be carried out in the custom renderer, so no additional implementation is required in the `MyEntry` control.

## Consuming the Custom Control

The `MyEntry` control can be referenced in XAML in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the control element. The following code example shows how the `MyEntry` control can be consumed by a XAML page:

```
<ContentPage ...>
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared the prefix is used to reference the custom control.

The following code example shows how the `MyEntry` control can be consumed by a C# page:

```
public class MainPage : ContentPage
{
    public MainPage ()
    {
        Content = new StackLayout {
            Children = {
                new Label {
                    Text = "Hello, Custom Renderer !",
                },
                new MyEntry {
                    Text = "In Shared Code",
                },
            },
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.CenterAndExpand,
        };
    }
}
```

This code instantiates a new `ContentPage` object that will display a `Label` and `MyEntry` control centered both vertically and horizontally on the page.

A custom renderer can now be added to each application project to customize the control's appearance on each platform.

## Creating the Custom Renderer on each Platform

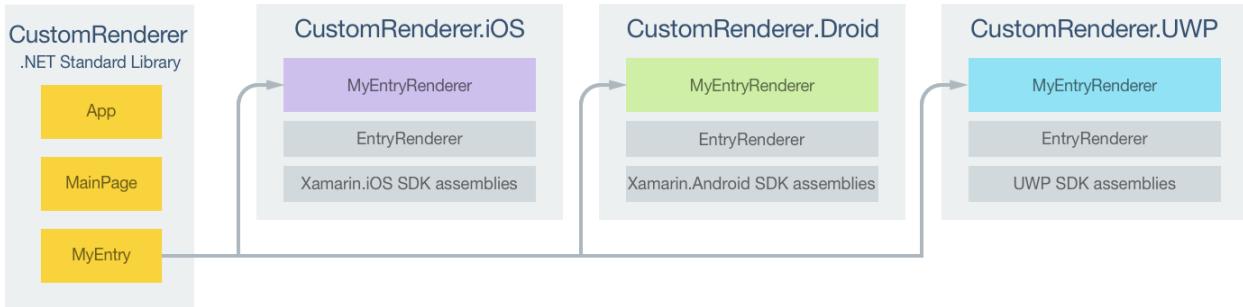
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `EntryRenderer` class that renders the native control.
2. Override the `OnElementChanged` method that renders the native control and write logic to customize the control.  
This method is called when the corresponding Xamarin.Forms control is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms control. This attribute is used to register the custom renderer with Xamarin.Forms.

## NOTE

It is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used.

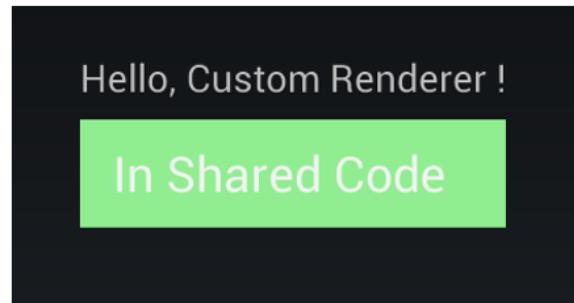
The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `MyEntry` control is rendered by platform-specific `MyEntryRenderer` classes, which all derive from the `EntryRenderer` class for each platform. This results in each `MyEntry` control being rendered with a platform-specific background color, as shown in the following screenshots:



iOS



Android

The `EntryRenderer` class exposes the `OnElementChanged` method, which is called when the `Xamarin.Forms` control is created to render the corresponding native control. This method takes an `ElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the `Xamarin.Forms` element that the renderer was attached to, and the `Xamarin.Forms` element that the renderer is attached to, respectively. In the sample application the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `MyEntry` control.

An overridden version of the `OnElementChanged` method in the `MyEntryRenderer` class is the place to perform the native control customization. A typed reference to the native control being used on the platform can be accessed through the `Control` property. In addition, a reference to the `Xamarin.Forms` control that's being rendered can be obtained through the `Element` property, although it's not used in the sample application.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with `Xamarin.Forms`. The attribute takes two parameters – the type name of the `Xamarin.Forms` control being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific `MyEntryRenderer` custom renderer class.

### Creating the Custom Renderer on iOS

The following code example shows the custom renderer for the iOS platform:

```
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer (typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.iOS
{
    public class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged (e);

            if (Control != null)
                // do whatever you want to the UITextField here!
                Control.BackgroundColor = UIColor.FromRGB (204, 153, 255);
                Control.BorderStyle = UITextBorderStyle.Line;
        }
    }
}
```

The call to the base class's `OnElementChanged` method instantiates an iOS `UITextField` control, with a reference to the control being assigned to the renderer's `Control` property. The background color is then set to light purple with the `UIColor.FromRGB` method.

## Creating the Custom Renderer on Android

The following code example shows the custom renderer for the Android platform:

```
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.Android
{
    class MyEntryRenderer : EntryRenderer
    {
        public MyEntryRenderer(Context context) : base(context)
        {

        }

        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);

            if (Control != null)
            {
                Control.SetBackgroundColor(global::Android.Graphics.Color.LightGreen);
            }
        }
    }
}
```

The call to the base class's `OnElementChanged` method instantiates an Android `EditText` control, with a reference to the control being assigned to the renderer's `Control` property. The background color is then set to light green with the `Control.SetBackgroundColor` method.

## Creating the Custom Renderer on UWP

The following code example shows the custom renderer for UWP:

```
[assembly: ExportRenderer(typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.UWP
{
    public class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);

            if (Control != null)
            {
                Control.Background = new SolidColorBrush(Colors.Cyan);
            }
        }
    }
}
```

The call to the base class's `OnElementChanged` method instantiates a `TextBox` control, with a reference to the control being assigned to the renderer's `Control` property. The background color is then set to cyan by creating a `SolidColorBrush` instance.

## Summary

This article has demonstrated how to create a custom control renderer for the `Xamarin.Forms Entry` control, enabling developers to override the default native rendering with their own platform-specific rendering. Custom renderers provide a powerful approach to customizing the appearance of `Xamarin.Forms` controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization.

## Related Links

- [CustomRendererEntry \(sample\)](#)

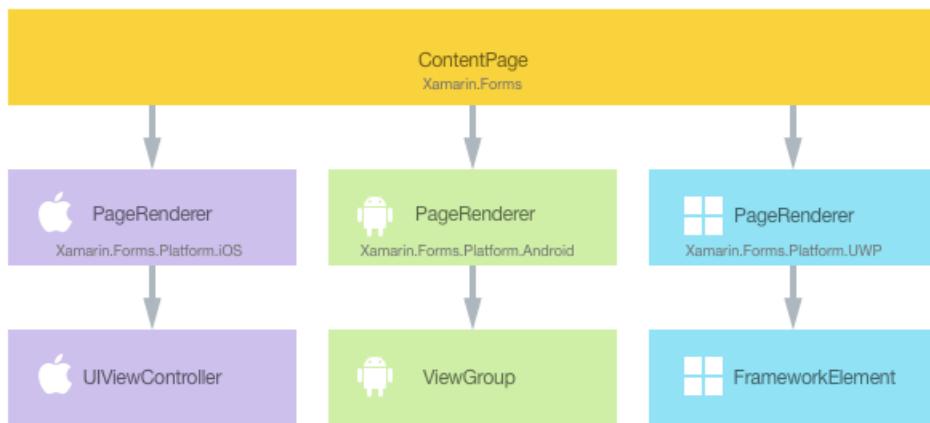
# Customizing a ContentPage

8/14/2018 • 7 minutes to read • [Edit Online](#)

A `ContentPage` is a visual element that displays a single view and occupies most of the screen. This article demonstrates how to create a custom renderer for the `ContentPage` page, enabling developers to override the default native rendering with their own platform-specific customization.

Every `Xamarin.Forms` control has an accompanying renderer for each platform that creates an instance of a native control. When a `ContentPage` is rendered by a `Xamarin.Forms` application, in iOS the `PageRenderer` class is instantiated, which in turn instantiates a native `UIViewController` control. On the Android platform, the `PageRenderer` class instantiates a `ViewGroup` control. On the Universal Windows Platform (UWP), the `PageRenderer` class instantiates a `FrameworkElement` control. For more information about the renderer and native control classes that `Xamarin.Forms` controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `ContentPage` and the corresponding native controls that implement it:



The rendering process can be taken advantage of to implement platform-specific customizations by creating a custom renderer for a `ContentPage` on each platform. The process for doing this is as follows:

1. [Create](#) a `Xamarin.Forms` page.
2. [Consume](#) the page from `Xamarin.Forms`.
3. [Create](#) the custom renderer for the page on each platform.

Each item will now be discussed in turn, to implement a `CameraPage` that provides a live camera feed and the ability to capture a photo.

## Creating the `Xamarin.Forms` Page

An unaltered `ContentPage` can be added to the shared `Xamarin.Forms` project, as shown in the following XAML code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"  
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"  
    x:Class="CustomRenderer.CameraPage">  
    <ContentPage.Content>  
        </ContentPage.Content>  
</ContentPage>
```

Similarly, the code-behind file for the `ContentPage` should also remain unaltered, as shown in the following code example:

```
public partial class CameraPage : ContentPage
{
    public CameraPage ()
    {
        // A custom renderer is used to display the camera UI
        InitializeComponent ();
    }
}
```

The following code example shows how the page can be created in C#:

```
public class CameraPageCS : ContentPage
{
    public CameraPageCS ()
    {
    }
}
```

An instance of the `CameraPage` will be used to display the live camera feed on each platform. Customization of the control will be carried out in the custom renderer, so no additional implementation is required in the `CameraPage` class.

## Consuming the Xamarin.Forms Page

The empty `CameraPage` must be displayed by the Xamarin.Forms application. This occurs when a button on the `MainPage` instance is tapped, which in turn executes the `OnTakePhotoButtonClicked` method, as shown in the following code example:

```
async void OnTakePhotoButtonClicked (object sender, EventArgs e)
{
    await Navigation.PushAsync (new CameraPage ());
}
```

This code simply navigates to the `CameraPage`, on which custom renderers will customize the page's appearance on each platform.

## Creating the Page Renderer on each Platform

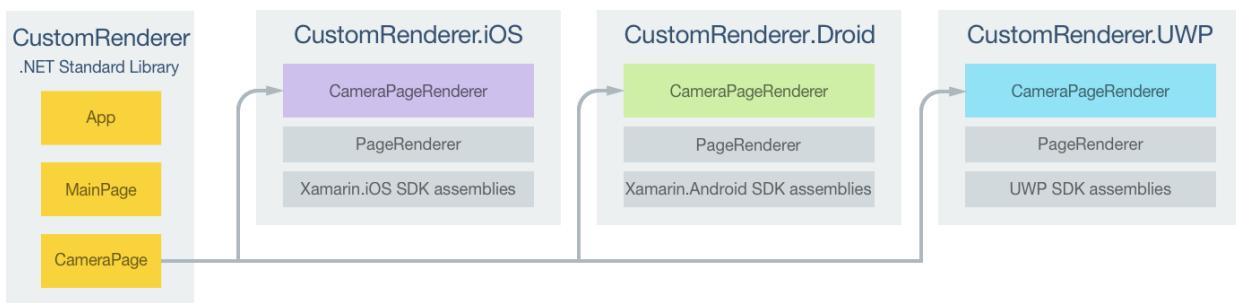
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `PageRenderer` class.
2. Override the `OnElementChanged` method that renders the native page and write logic to customize the page. The `OnElementChanged` method is called when the corresponding Xamarin.Forms control is created.
3. Add an `ExportRenderer` attribute to the page renderer class to specify that it will be used to render the Xamarin.Forms page. This attribute is used to register the custom renderer with Xamarin.Forms.

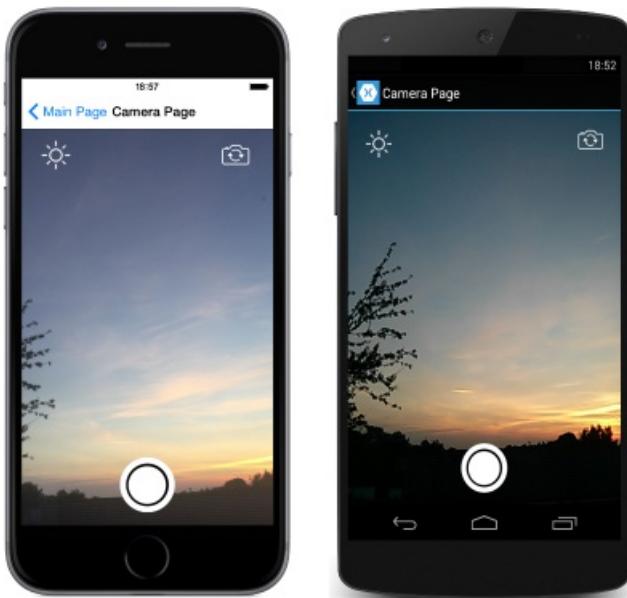
### NOTE

It is optional to provide a page renderer in each platform project. If a page renderer isn't registered, then the default renderer for the page will be used.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationship between them:



The `CameraPage` instance is rendered by platform-specific `CameraPageRenderer` classes, which all derive from the `PageRenderer` class for that platform. This results in each `CameraPage` instance being rendered with a live camera feed, as shown in the following screenshots:



The `PageRenderer` class exposes the `OnElementChanged` method, which is called when the Xamarin.Forms page is created to render the corresponding native control. This method takes an `ElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the Xamarin.Forms element that the renderer was attached to, and the Xamarin.Forms element that the renderer is attached to, respectively. In the sample application the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `CameraPage` instance.

An overridden version of the `OnElementChanged` method in the `CameraPageRenderer` class is the place to perform the native page customization. A reference to the Xamarin.Forms page instance that's being rendered can be obtained through the `Element` property.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms page being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of the `CameraPageRenderer` custom renderer for each platform.

### Creating the Page Renderer on iOS

The following code example shows the page renderer for the iOS platform:

```
[assembly:ExportRenderer (typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.iOS
{
    public class CameraPageRenderer : PageRenderer
    {
        ...
        protected override void OnElementChanged (VisualElementChangedEventArgs e)
        {
            base.OnElementChanged (e);

            if (e.OldElement != null || Element == null) {
                return;
            }

            try {
                SetupUserInterface ();
                SetupEventHandlers ();
                SetupLiveCameraStream ();
                AuthorizeCameraUse ();
            } catch (Exception ex) {
                System.Diagnostics.Debug.WriteLine ("@"
                    ERROR: ", ex.Message);
            }
        }
        ...
    }
}
```

The call to the base class's `OnElementChanged` method instantiates an iOS `UIViewController` control. The live camera stream is only rendered provided that the renderer isn't already attached to an existing Xamarin.Forms element, and provided that a page instance exists that is being rendered by the custom renderer.

The page is then customized by a series of methods that use the `AVCapture` APIs to provide the live stream from the camera and the ability to capture a photo.

### **Creating the Page Renderer on Android**

The following code example shows the page renderer for the Android platform:

```

[assembly: ExportRenderer(typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.Droid
{
    public class CameraPageRenderer : PageRenderer, TextureView.ISurfaceTextureListener
    {
        ...
        public CameraPageRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Page> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            try
            {
                SetupUserInterface();
                SetupEventHandlers();
                AddView(view);
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(@"
                    ERROR: ", ex.Message);
            }
        }
        ...
    }
}

```

The call to the base class's `OnElementChanged` method instantiates an Android `ViewGroup` control, which is a group of views. The live camera stream is only rendered provided that the renderer isn't already attached to an existing Xamarin.Forms element, and provided that a page instance exists that is being rendered by the custom renderer.

The page is then customized by invoking a series of methods that use the `Camera` API to provide the live stream from the camera and the ability to capture a photo, before the `AddView` method is invoked to add the live camera stream UI to the `ViewGroup`. Note that on Android it's also necessary to override the `onLayout` method to perform measure and layout operations on the view. For more information, see the [ContentPage renderer sample](#).

## Creating the Page Renderer on UWP

The following code example shows the page renderer for UWP:

```
[assembly: ExportRenderer(typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.UWP
{
    public class CameraPageRenderer : PageRenderer
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.Page> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            try
            {
                ...
                SetupUserInterface();
                SetupBasedOnStateAsync();

                this.Children.Add(page);
            }
            ...
        }

        protected override Size ArrangeOverride(Size finalSize)
        {
            page.Arrange(new Windows.Foundation.Rect(0, 0, finalSize.Width, finalSize.Height));
            return finalSize;
        }
        ...
    }
}
```

The call to the base class's `OnElementChanged` method instantiates a `FrameworkElement` control, on which the page is rendered. The live camera stream is only rendered provided that the renderer isn't already attached to an existing `Xamarin.Forms` element, and provided that a page instance exists that is being rendered by the custom renderer. The page is then customized by invoking a series of methods that use the `MediaCapture` API to provide the live stream from the camera and the ability to capture a photo before the customized page is added to the `Children` collection for display.

When implementing a custom renderer that derives from `PageRenderer` on UWP, the `ArrangeOverride` method should also be implemented to arrange the page controls, because the base renderer doesn't know what to do with them. Otherwise, a blank page results. Therefore, in this example the `ArrangeOverride` method calls the `Arrange` method on the `Page` instance.

#### NOTE

It's important to stop and dispose of the objects that provide access to the camera in a UWP application. Failure to do so can interfere with other applications that attempt to access the device's camera. For more information, see [Display the camera preview](#).

## Summary

This article has demonstrated how to create a custom renderer for the `ContentPage` page, enabling developers to override the default native rendering with their own platform-specific customization. A `ContentPage` is a visual element that displays a single view and occupies most of the screen.

## Related Links

- [CustomRendererContentPage \(sample\)](#)

# Customizing a Xamarin.Forms Map

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms.Maps provides a cross-platform abstraction for displaying maps that use the native map APIs on each platform, to provide a fast and familiar map experience for users.*

## Customizing a Map Pin

This article explains how to create a custom renderer for the `Map` control, which displays a native map with a customized pin and a customized view of the pin data on each platform.

## Highlighting a Circular Area on a Map

This article explains how to add a circular overlay to a map, to highlight a circular area of the map.

## Highlighting a Region on a Map

This article explains how to add a polygon overlay to a map, to highlight a region on the map. Polygons are a closed shape and have their interiors filled in.

## Highlighting a Route on a Map

This article explains how to add a polyline overlay to a map. A polyline overlay is a series of connected line segments that are typically used to show a route on a map, or form any shape that's required.

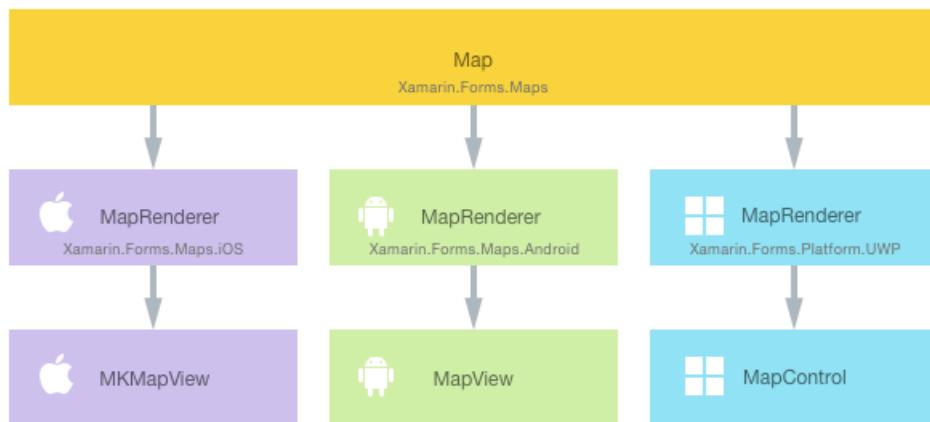
# Customizing a Map Pin

11/20/2018 • 20 minutes to read • [Edit Online](#)

This article demonstrates how to create a custom renderer for the `Map` control, which displays a native map with a customized pin and a customized view of the pin data on each platform.

Every Xamarin.Forms view has an accompanying renderer for each platform that creates an instance of a native control. When a `Map` is rendered by a Xamarin.Forms application in iOS, the `MapRenderer` class is instantiated, which in turn instantiates a native `MKMapView` control. On the Android platform, the `MapRenderer` class instantiates a native `MapView` control. On the Universal Windows Platform (UWP), the `MapRenderer` class instantiates a native `MapControl`. For more information about the renderer and native control classes that Xamarin.Forms controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `Map` and the corresponding native controls that implement it:



The rendering process can be used to implement platform-specific customizations by creating a custom renderer for a `Map` on each platform. The process for doing this is as follows:

1. [Create](#) a Xamarin.Forms custom map.
2. [Consume](#) the custom map from Xamarin.Forms.
3. [Create](#) the custom renderer for the map on each platform.

Each item will now be discussed in turn, to implement a `CustomMap` renderer that displays a native map with a customized pin and a customized view of the pin data on each platform.

## NOTE

`Xamarin.Forms.Maps` must be initialized and configured before use. For more information, see [Maps Control](#).

## Creating the Custom Map

A custom map control can be created by subclassing the `Map` class, as shown in the following code example:

```
public class CustomMap : Map
{
    public List<CustomPin> CustomPins { get; set; }
}
```

The `CustomMap` control is created in the .NET Standard library project and defines the API for the custom map. The custom map exposes the `CustomPins` property that represents the collection of `CustomPin` objects that will be rendered by the native map control on each platform. The `CustomPin` class is shown in the following code example:

```
public class CustomPin : Pin
{
    public string Url { get; set; }
}
```

This class defines a `CustomPin` as inheriting the properties of the `Pin` class, and adding a `Url` property.

## Consuming the Custom Map

The `CustomMap` control can be referenced in XAML in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the custom map control. The following code example shows how the `CustomMap` control can be consumed by a XAML page:

```
<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer">
<ContentPage.Content>
    <local:CustomMap x:Name="myMap" MapType="Street"
        WidthRequest="{x:Static local:App.ScreenWidth}"
        HeightRequest="{x:Static local:App.ScreenHeight}" />
</ContentPage.Content>
</ContentPage>
```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom map. Once the namespace is declared, the prefix is used to reference the custom map.

The following code example shows how the `CustomMap` control can be consumed by a C# page:

```
public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...

        Content = customMap;
    }
}
```

The `CustomMap` instance will be used to display the native map on each platform. Its `MapType` property sets the display style of the `Map`, with the possible values being defined in the `MapType` enumeration. For iOS and Android, the width and height of the map is set through properties of the `App` class that are initialized in the

platform-specific projects.

The location of the map, and the pins it contains, are initialized as shown in the following code example:

```
public MapPage ()  
{  
    ...  
    var pin = new CustomPin {  
        Type = PinType.Place,  
        Position = new Position (37.79752, -122.40183),  
        Label = "Xamarin San Francisco Office",  
        Address = "394 Pacific Ave, San Francisco CA",  
        Id = "Xamarin",  
        Url = "http://xamarin.com/about/"  
    };  
  
    customMap.CustomPins = new List<CustomPin> { pin };  
    customMap.Pins.Add (pin);  
    customMap.MoveToRegion (MapSpan.FromCenterAndRadius (  
        new Position (37.79752, -122.40183), Distance.FromMiles (1.0)));  
}
```

This initialization adds a custom pin and positions the map's view with the `MoveToRegion` method, which changes the position and zoom level of the map by creating a `MapSpan` from a `Position` and a `Distance`.

A custom renderer can now be added to each application project to customize the native map controls.

## Creating the Custom Renderer on each Platform

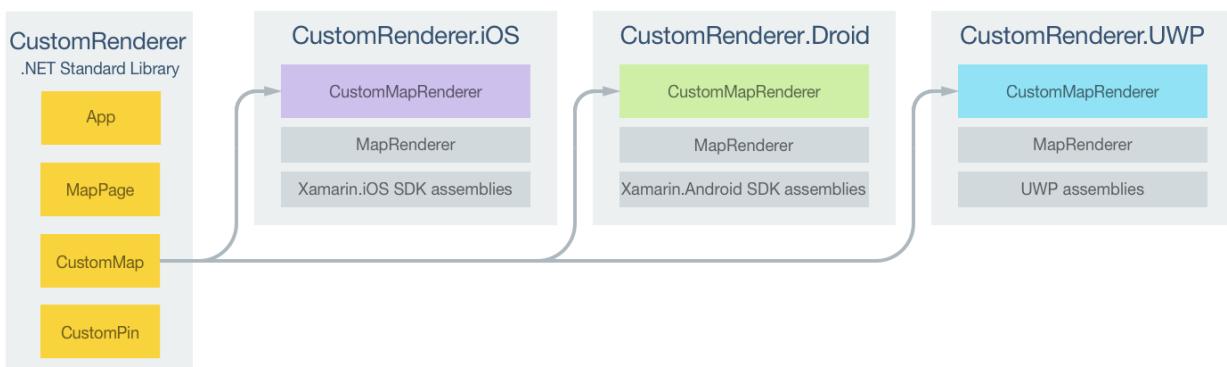
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `MapRenderer` class that renders the custom map.
2. Override the `OnElementChanged` method that renders the custom map and write logic to customize it. This method is called when the corresponding Xamarin.Forms custom map is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms custom map. This attribute is used to register the custom renderer with Xamarin.Forms.

### NOTE

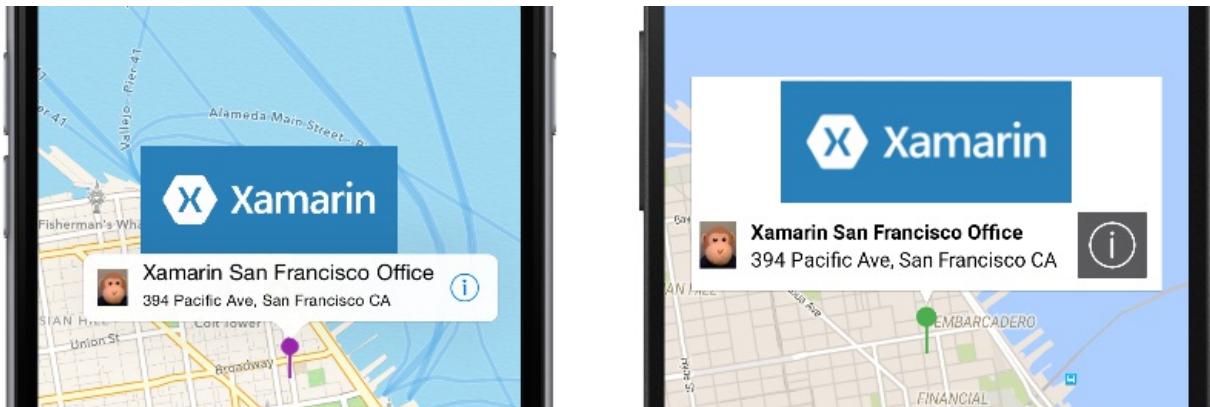
It is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `CustomMap` control is rendered by platform-specific renderer classes, which derive from the `MapRenderer` class

for each platform. This results in each `CustomMap` control being rendered with platform-specific controls, as shown in the following screenshots:



The `MapRenderer` class exposes the `OnElementChanged` method, which is called when the `Xamarin.Forms` custom map is created to render the corresponding native control. This method takes an `ElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the `Xamarin.Forms` element that the renderer was attached to, and the `Xamarin.Forms` element that the renderer is attached to, respectively. In the sample application the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `CustomMap` instance.

An overridden version of the `OnElementChanged` method, in each platform-specific renderer class, is the place to perform the native control customization. A typed reference to the native control being used on the platform can be accessed through the `Control` property. In addition, a reference to the `Xamarin.Forms` control that's being rendered can be obtained through the `Element` property.

Care must be taken when subscribing to event handlers in the `OnElementChanged` method, as demonstrated in the following code example:

```
protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
{
    base.OnElementChanged (e);

    if (e.OldElement != null) {
        // Unsubscribe from event handlers
    }

    if (e.NewElement != null) {
        // Configure the native control and subscribe to event handlers
    }
}
```

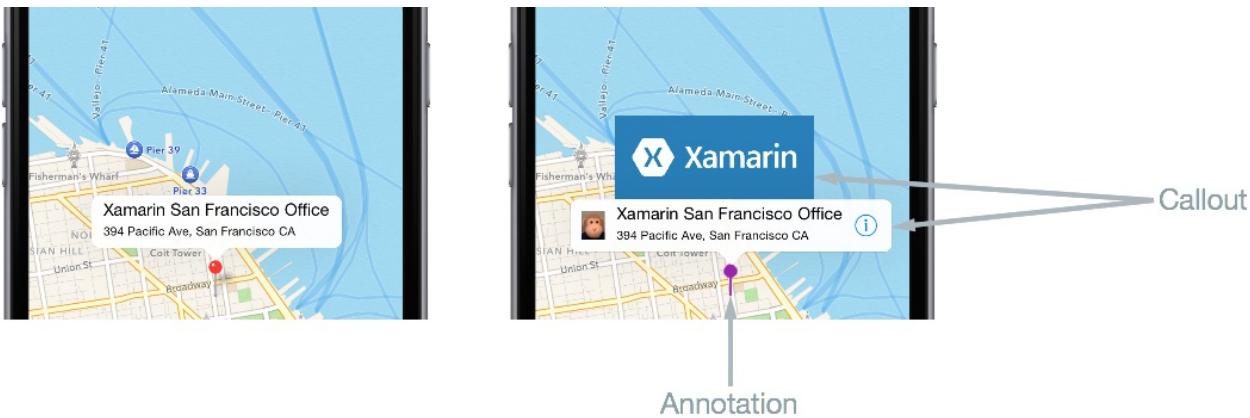
The native control should be configured and event handlers subscribed to only when the custom renderer is attached to a new `Xamarin.Forms` element. Similarly, any event handlers that were subscribed to should be unsubscribed from only when the element that the renderer is attached to changes. Adopting this approach will help to create a custom renderer that doesn't suffer from memory leaks.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with `Xamarin.Forms`. The attribute takes two parameters – the type name of the `Xamarin.Forms` custom control being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific custom renderer class.

## Creating the Custom Renderer on iOS

The following screenshots show the map, before and after customization:



On iOS the pin is called an *annotation*, and can be either a custom image or a system-defined pin of various colors. Annotations can optionally show a *callout*, which is displayed in response to the user selecting the annotation. The callout displays the `Label` and `Address` properties of the `Pin` instance, with optional left and right accessory views. In the screenshot above, the left accessory view is the image of a monkey, with the right accessory view being the *Information* button.

The following code example shows the custom renderer for the iOS platform:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        UIView customPinView;
        List<CustomPin> customPins;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveAnnotations(nativeMap.Annotations);
                    nativeMap.GetViewForAnnotation = null;
                    nativeMap.CalloutAccessoryControlTapped -= OnCalloutAccessoryControlTapped;
                    nativeMap.DidSelectAnnotationView -= OnDidSelectAnnotationView;
                    nativeMap.DidDeselectAnnotationView -= OnDidDeselectAnnotationView;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                customPins = formsMap.CustomPins;

                nativeMap.GetViewForAnnotation = GetViewForAnnotation;
                nativeMap.CalloutAccessoryControlTapped += OnCalloutAccessoryControlTapped;
                nativeMap.DidSelectAnnotationView += OnDidSelectAnnotationView;
                nativeMap.DidDeselectAnnotationView += OnDidDeselectAnnotationView;
            }
        }
        ...
    }
}
```

The `OnElementChanged` method performs the following configuration of the `MKMapView` instance, provided that the custom renderer is attached to a new `Xamarin.Forms` element:

- The `GetViewForAnnotation` property is set to the `GetViewForAnnotation` method. This method is called when the

location of the annotation becomes visible on the map, and is used to customize the annotation prior to display.

- Event handlers for the `CalloutAccessoryControlTapped`, `DidSelectAnnotationView`, and `DidDeselectAnnotationView` events are registered. These events fire when the user [taps the right accessory in the callout](#), and when the user [selects](#) and [deselects](#) the annotation, respectively. The events are unsubscribed from only when the element the renderer is attached to changes.

### Displaying the Annotation

The `GetViewForAnnotation` method is called when the location of the annotation becomes visible on the map, and is used to customize the annotation prior to display. An annotation has two parts:

- `MkAnnotation` – includes the title, subtitle, and location of the annotation.
- `MkAnnotationView` – contains the image to represent the annotation, and optionally, a callout that is shown when the user taps the annotation.

The `GetViewForAnnotation` method accepts an `IMKAnnotation` that contains the annotation's data and returns an `MKAnnotationView` for display on the map, and is shown in the following code example:

```
protected override MKAnnotationView GetViewForAnnotation(MKMapView mapView, IMKAnnotation annotation)
{
    MKAnnotationView annotationView = null;

    if (annotation is MKUserLocation)
        return null;

    var customPin = GetCustomPin(annotation as MKPointAnnotation);
    if (customPin == null) {
        throw new Exception("Custom pin not found");
    }

    annotationView = mapView.DequeueReusableAnnotation(customPin.Id.ToString());
    if (annotationView == null) {
        annotationView = new CustomMKAnnotationView(annotation, customPin.Id.ToString());
        annotationView.Image = UIImage.FromFile("pin.png");
        annotationView.CalloutOffset = new CGPoint(0, 0);
        annotationView.LeftCalloutAccessoryView = new UIImageView(UIImage.FromFile("monkey.png"));
        annotationView.RightCalloutAccessoryView = UIButton.FromType(UIButtonType.DetailDisclosure);
        ((CustomMKAnnotationView)annotationView).Id = customPin.Id.ToString();
        ((CustomMKAnnotationView)annotationView).Url = customPin.Url;
    }
    annotationView.CanShowCallout = true;

    return annotationView;
}
```

This method ensures that the annotation will be displayed as a custom image, rather than as system-defined pin, and that when the annotation is tapped a callout will be displayed that includes additional content to the left and right of the annotation title and address. This is accomplished as follows:

- The `GetCustomPin` method is called to return the custom pin data for the annotation.
- To conserve memory, the annotation's view is pooled for reuse with the call to `DequeueReusableAnnotation`.
- The `CustomMKAnnotationView` class extends the `MKAnnotationView` class with `Id` and `Url` properties that correspond to identical properties in the `CustomPin` instance. A new instance of the `CustomMKAnnotationView` is created, provided that the annotation is `null`:
  - The `CustomMKAnnotationView.Image` property is set to the image that will represent the annotation on the map.
  - The `CustomMKAnnotationView.CalloutOffset` property is set to a `CGPoint` that specifies that the callout will be centered above the annotation.
  - The `CustomMKAnnotationView.LeftCalloutAccessoryView` property is set to an image of a monkey that will

appear to the left of the annotation title and address.

- The `CustomMKAnnotationView.RightCalloutAccessoryView` property is set to an *Information* button that will appear to the right of the annotation title and address.
  - The `CustomMKAnnotationView.Id` property is set to the `CustomPin.Id` property returned by the `GetCustomPin` method. This enables the annotation to be identified so that its [callout can be further customized](#), if desired.
  - The `CustomMKAnnotationView.Url` property is set to the `CustomPin.Url` property returned by the `GetCustomPin` method. The URL will be navigated to when the user [taps the button displayed in the right callout accessory view](#).
4. The `MKAnnotationView.CanShowCallout` property is set to `true` so that the callout is displayed when the annotation is tapped.
  5. The annotation is returned for display on the map.

### Selecting the Annotation

When the user taps on the annotation, the `DidSelectAnnotationView` event fires, which in turn executes the `OnDidSelectAnnotationView` method:

```
void OnDidSelectAnnotationView (object sender, MKAnnotationViewEventArgs e)
{
    var customView = e.View as CustomMKAnnotationView;
    customPinView = new UIView ();

    if (customView.Id == "Xamarin") {
        customPinView.Frame = new CGRect (0, 0, 200, 84);
        var image = new UIImageView (new CGRect (0, 0, 200, 84));
        image.Image = UIImage.FromFile ("xamarin.png");
        customPinView.AddSubview (image);
        customPinView.Center = new CGPoint (0, -(e.View.Frame.Height + 75));
        e.View.AddSubview (customPinView);
    }
}
```

This method extends the existing callout (that contains left and right accessory views) by adding a `UIView` instance to it that contains an image of the Xamarin logo, provided that the selected annotation has its `Id` property set to `Xamarin`. This allows for scenarios where different callouts can be displayed for different annotations. The `UIView` instance will be displayed centered above the existing callout.

### Tapping on the Right Callout Accessory View

When the user taps on the *Information* button in the right callout accessory view, the `CalloutAccessoryControlTapped` event fires, which in turn executes the `OnCalloutAccessoryControlTapped` method:

```
void OnCalloutAccessoryControlTapped (object sender, MKMapViewAccessoryTappedEventArgs e)
{
    var customView = e.View as CustomMKAnnotationView;
    if (!string.IsNullOrWhiteSpace (customView.Url)) {
        UIApplication.SharedApplication.OpenUrl (new Foundation.NSUrl (customView.Url));
    }
}
```

This method opens a web browser and navigates to the address stored in the `CustomMKAnnotationView.Url` property. Note that the address was defined when creating the `CustomPin` collection in the .NET Standard library project.

### Deselecting the Annotation

When the annotation is displayed and the user taps on the map, the `DidDeselectAnnotationView` event fires, which in turn executes the `OnDidDeselectAnnotationView` method:

```

void OnDidDeselectAnnotationView (object sender, MKAnnotationEventArgs e)
{
    if (!e.View.Selected) {
        customPinView.RemoveFromSuperview ();
        customPinView.Dispose ();
        customPinView = null;
    }
}

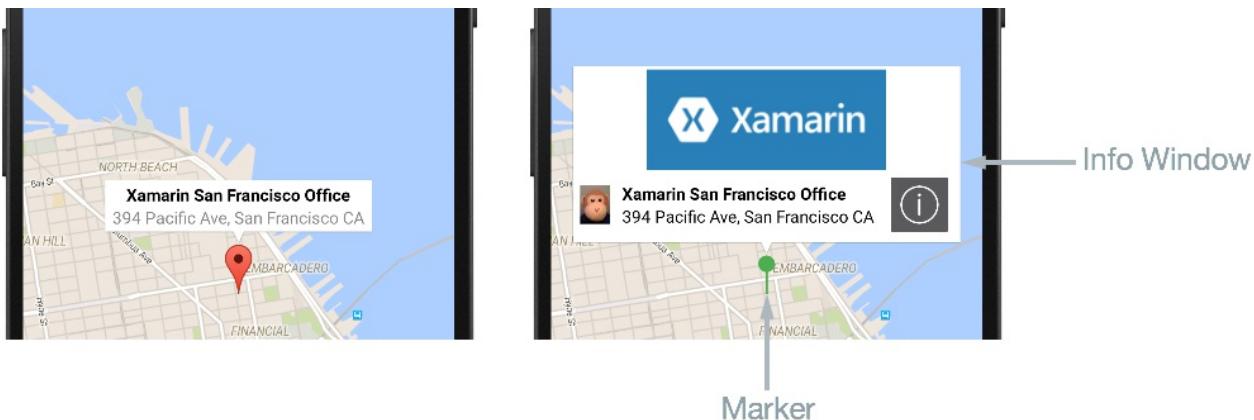
```

This method ensures that when the existing callout is not selected, the extended part of the callout (the image of the Xamarin logo) will also stop being displayed, and its resources will be released.

For more information about customizing a `MKMapView` instance, see [iOS Maps](#).

### Creating the Custom Renderer on Android

The following screenshots show the map, before and after customization:



On Android the pin is called a *marker*, and can either be a custom image or a system-defined marker of various colors. Markers can show an *info window*, which is displayed in response to the user tapping on the marker. The info window displays the `Label` and `Address` properties of the `Pin` instance, and can be customized to include other content. However, only one info window can be shown at once.

The following code example shows the custom renderer for the Android platform:

```

[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.Droid
{
    public class CustomMapRenderer : MapRenderer, GoogleMap.IInfoWindowAdapter
    {
        List<CustomPin> customPins;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                NativeMap.InfoWindowClick -= OnInfoWindowClick;
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                customPins = formsMap.CustomPins;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(GoogleMap map)
        {
            base.OnMapReady(map);

            NativeMap.InfoWindowClick += OnInfoWindowClick;
            NativeMap.SetInfoWindowAdapter(this);
        }
        ...
    }
}

```

Provided that the custom renderer is attached to a new Xamarin.Forms element, the `OnElementChanged` method calls the `MapView.GetMapAsync` method, which gets the underlying `GoogleMap` that is tied to the view. Once the `GoogleMap` instance is available, the `OnMapReady` override will be invoked. This method registers an event handler for the `InfoWindowClick` event, which fires when the [info window is clicked](#), and is unsubscribed from only when the element the renderer is attached to changes. The `OnMapReady` override also calls the `SetInfoWindowAdapter` method to specify that the `CustomMapRenderer` class instance will provide the methods to customize the info window.

The `CustomMapRenderer` class implements the `GoogleMap.IInfoWindowAdapter` interface to [customize the info window](#). This interface specifies that the following methods must be implemented:

- `public Android.Views.View GetInfoWindow(Marker marker)` – This method is called to return a custom info window for a marker. If it returns `null`, then the default window rendering will be used. If it returns a `View`, then that `View` will be placed inside the info window frame.
- `public Android.Views.View GetInfoContents(Marker marker)` – This method is called to return a `View` containing the content of the info window, and will only be called if the `GetInfoWindow` method returns `null`. If it returns `null`, then the default rendering of the info window content will be used.

In the sample application, only the info window content is customized, and so the `GetInfoWindow` method returns `null` to enable this.

## Customizing the Marker

The icon used to represent a marker can be customized by calling the `MarkerOptions.SetIcon` method. This can be accomplished by overriding the `CreateMarker` method, which is invoked for each `Pin` that's added to the map:

```
protected override MarkerOptions CreateMarker(Pin pin)
{
    var marker = new MarkerOptions();
    marker.SetPosition(new LatLng(pin.Position.Latitude, pin.Position.Longitude));
    markerSetTitle(pin.Label);
    marker.SetSnippet(pin.Address);
    marker.SetIcon(BitmapDescriptorFactory.FromResource(Resource.Drawable.pin));
    return marker;
}
```

This method creates a new `MarkerOption` instance for each `Pin` instance. After setting the position, label, and address of the marker, its icon is set with the `SetIcon` method. This method takes a `BitmapDescriptor` object containing the data necessary to render the icon, with the `BitmapDescriptorFactory` class providing helper methods to simplify the creation of the `BitmapDescriptor`. For more information about using the `BitmapDescriptorFactory` class to customize a marker, see [Customizing a Marker](#).

### NOTE

If required, the `GetMarkerForPin` method can be invoked in your map renderer to retrieve a `Marker` from a `Pin`.

## Customizing the Info Window

When a user taps on the marker, the `GetInfoContents` method is executed, provided that the `GetInfoWindow` method returns `null`. The following code example shows the `GetInfoContents` method:

```
public Android.Views.View GetInfoContents (Marker marker)
{
    var inflater = Android.App.Application.Context.GetService (Context.LayoutInflaterService) as
Android.Views.LayoutInflater;
    if (inflater != null) {
        Android.Views.View view;

        var customPin = GetCustomPin (marker);
        if (customPin == null) {
            throw new Exception ("Custom pin not found");
        }

        if (customPin.Id.ToString() == "Xamarin") {
            view = inflater.Inflate (Resource.Layout.XamarinMapInfoWindow, null);
        } else {
            view = inflater.Inflate (Resource.Layout.MapInfoWindow, null);
        }

        var infoTitle = view.FindViewById<TextView> (Resource.Id.InfoWindowTitle);
        var infoSubtitle = view.FindViewById<TextView> (Resource.Id.InfoWindowSubtitle);

        if (infoTitle != null) {
            infoTitle.Text = marker.Title;
        }
        if (infoSubtitle != null) {
            infoSubtitle.Text = marker.Snippet;
        }

        return view;
    }
    return null;
}
```

This method returns a `View` containing the contents of the info window. This is accomplished as follows:

- A `LayoutInflater` instance is retrieved. This is used to instantiate a layout XML file into its corresponding `View`.
- The `GetCustomPin` method is called to return the custom pin data for the info window.
- The `XamarinMapInfoWindow` layout is inflated if the `CustomPin.Id` property is equal to `Xamarin`. Otherwise, the `MapInfoWindow` layout is inflated. This allows for scenarios where different info window layouts can be displayed for different markers.
- The `InfoWindowTitle` and `InfoWindowSubtitle` resources are retrieved from the inflated layout, and their `Text` properties are set to the corresponding data from the `Marker` instance, provided that the resources are not `null`.
- The `View` instance is returned for display on the map.

#### NOTE

An info window is not a live `View`. Instead, Android will convert the `View` to a static bitmap and display that as an image. This means that while an info window can respond to a click event, it cannot respond to any touch events or gestures, and the individual controls in the info window cannot respond to their own click events.

#### Clicking on the Info Window

When the user clicks on the info window, the `InfoWindowClick` event fires, which in turn executes the `OnInfoWindowClick` method:

```
void OnInfoWindowClick (object sender, GoogleMap.InfoWindowEventArgs e)
{
    var customPin = GetCustomPin (e.Marker);
    if (customPin == null) {
        throw new Exception ("Custom pin not found");
    }

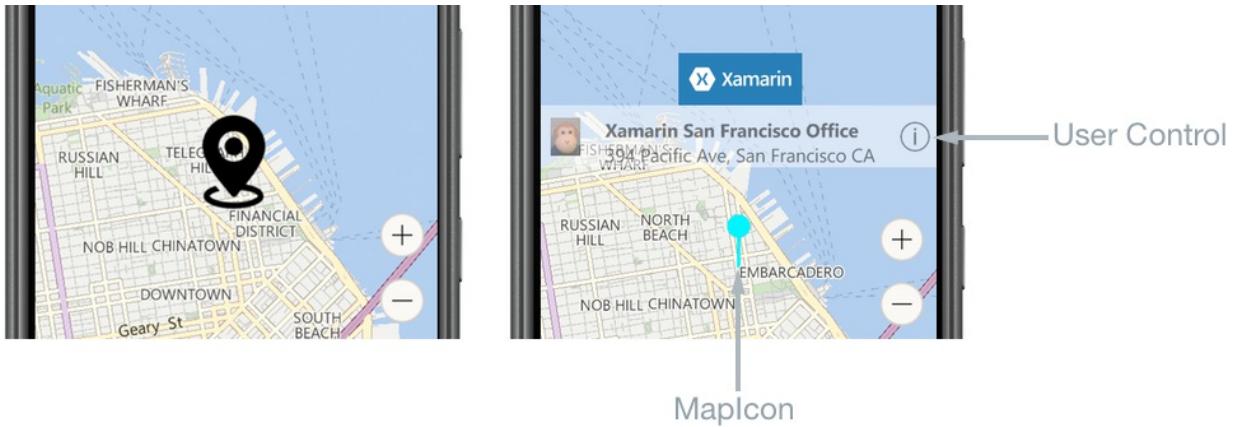
    if (!string.IsNullOrWhiteSpace (customPin.Url)) {
        var url = Android.Net.Uri.Parse (customPin.Url);
        var intent = new Intent (Intent.ActionView, url);
        intent.AddFlags (ActivityFlags.NewTask);
        Android.App.Application.Context.StartActivity (intent);
    }
}
```

This method opens a web browser and navigates to the address stored in the `Url` property of the retrieved `CustomPin` instance for the `Marker`. Note that the address was defined when creating the `CustomPin` collection in the .NET Standard library project.

For more information about customizing a `MapView` instance, see [Maps API](#).

#### Creating the Custom Renderer on the Universal Windows Platform

The following screenshots show the map, before and after customization:



On UWP the pin is called a *map icon*, and can either be a custom image or the system-defined default image. A map icon can show a `UserControl`, which is displayed in response to the user tapping on the map icon. The `UserControl` can display any content, including the `Label` and `Address` properties of the `Pin` instance.

The following code example shows the UWP custom renderer:

```

[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        MapControl nativeMap;
        List<CustomPin> customPins;
        XamarinMapOverlay mapOverlay;
        bool xamarinOverlayShown = false;

        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                nativeMap.MapElementClick -= OnMapElementClick;
                nativeMap.Children.Clear();
                mapOverlay = null;
                nativeMap = null;
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                nativeMap = Control as MapControl;
                customPins = formsMap.CustomPins;

                nativeMap.Children.Clear();
                nativeMap.MapElementClick += OnMapElementClick;

                foreach (var pin in customPins)
                {
                    var snPosition = new BasicGeoposition { Latitude = pin.Pin.Position.Latitude, Longitude = pin.Pin.Position.Longitude };
                    var snPoint = new Geopoint(snPosition);

                    var mapIcon = new MapIcon();
                    mapIcon.Image = RandomAccessStreamReference.CreateFromUri(new Uri("ms-appx:///pin.png"));
                    mapIcon.CollisionBehaviorDesired = MapElementCollisionBehavior.RemainVisible;
                    mapIcon.Location = snPoint;
                    mapIcon.NormalizedAnchorPoint = new Windows.Foundation.Point(0.5, 1.0);

                    nativeMap.MapElements.Add(mapIcon);
                }
            }
        }
        ...
    }
}

```

The `OnElementChanged` method performs the following operations, provided that the custom renderer is attached to a new Xamarin.Forms element:

- It clears the `MapControl.Children` collection to remove any existing user interface elements from the map, before registering an event handler for the `MapElementclick` event. This event fires when the user taps or clicks on a `MapElement` on the `MapControl`, and is unsubscribed from only when the element the renderer is attached to changes.
- Each pin in the `customPins` collection is displayed at the correct geographic location on the map as follows:
  - The location for the pin is created as a `Geopoint` instance.
  - A `MapIcon` instance is created to represent the pin.
  - The image used to represent the `MapIcon` is specified by setting the `MapIcon.Image` property. However,

the map icon's image is not always guaranteed to be shown, as it may be obscured by other elements on the map. Therefore, the map icon's `CollisionBehaviorDesired` property is set to `MapElementCollisionBehavior.RemainVisible`, to ensure that it remains visible.

- The location of the `MapIcon` is specified by setting the `MapIcon.Location` property.
- The `MapIcon.NormalizedAnchorPoint` property is set to the approximate location of the pointer on the image. If this property retains its default value of (0,0), which represents the upper left corner of the image, changes in the zoom level of the map may result in the image pointing to a different location.
- The `MapIcon` instance is added to the `MapControl.MapElements` collection. This results in the map icon being displayed on the `MapControl`.

#### NOTE

When using the same image for multiple map icons, the `RandomAccessStreamReference` instance should be declared at the page or application level for best performance.

#### Displaying the UserControl

When a user taps on the map icon, the `OnMapElementClick` method is executed. The following code example shows this method:

```
private void OnMapElementClick(MapControl sender, MapElementEventArgs args)
{
    var mapIcon = args.MapElements.FirstOrDefault(x => x is MapIcon) as MapIcon;
    if (mapIcon != null)
    {
        if (!xamarinOverlayShown)
        {
            var customPin = GetCustomPin(mapIcon.Location.Position);
            if (customPin == null)
            {
                throw new Exception("Custom pin not found");
            }

            if (customPin.Id.ToString() == "Xamarin")
            {
                if (mapOverlay == null)
                {
                    mapOverlay = new XamarinMapOverlay(customPin);
                }

                var snPosition = new BasicGeoposition { Latitude = customPin.Pin.Position.Latitude, Longitude = customPin.Pin.Position.Longitude };
                var snPoint = new Geopoint(snPosition);

                nativeMap.Children.Add(mapOverlay);
                MapControl.SetLocation(mapOverlay, snPoint);
                MapControl.SetNormalizedAnchorPoint(mapOverlay, new Windows.Foundation.Point(0.5, 1.0));
                xamarinOverlayShown = true;
            }
        }
        else
        {
            nativeMap.Children.Remove(mapOverlay);
            xamarinOverlayShown = false;
        }
    }
}
```

This method creates a `UserControl` instance that displays information about the pin. This is accomplished as follows:

- The `MapIcon` instance is retrieved.
- The `GetCustomPin` method is called to return the custom pin data that will be displayed.
- A `XamarinMapOverlay` instance is created to display the custom pin data. This class is a user control.
- The geographic location at which to display the `XamarinMapOverlay` instance on the `MapControl` is created as a `Geopoint` instance.
- The `XamarinMapOverlay` instance is added to the `MapControl.Children` collection. This collection contains XAML user interface elements that will be displayed on the map.
- The geographic location of the `XamarinMapOverlay` instance on the map is set by calling the `SetLocation` method.
- The relative location on the `XamarinMapOverlay` instance, that corresponds to the specified location, is set by calling the `SetNormalizedAnchorPoint` method. This ensures that changes in the zoom level of the map result in the `XamarinMapOverlay` instance always being displayed at the correct location.

Alternatively, if information about the pin is already being displayed on the map, tapping on the map removes the `XamarinMapOverlay` instance from the `MapControl.Children` collection.

#### Tapping on the Information Button

When the user taps on the *Information* button in the `XamarinMapOverlay` user control, the `Tapped` event fires, which in turn executes the `OnInfoButtonTapped` method:

```
private async void OnInfoButtonTapped(object sender, TappedRoutedEventArgs e)
{
    await Launcher.LaunchUriAsync(new Uri(customPin.Url));
}
```

This method opens a web browser and navigates to the address stored in the `Url` property of the `CustomPin` instance. Note that the address was defined when creating the `CustomPin` collection in the .NET Standard library project.

For more information about customizing a `MapControl` instance, see [Maps and Location Overview](#) on MSDN.

## Summary

This article demonstrated how to create a custom renderer for the `Map` control, enabling developers to override the default native rendering with their own platform-specific customization. Xamarin.Forms.Maps provides a cross-platform abstraction for displaying maps that use the native map APIs on each platform to provide a fast and familiar map experience for users.

## Related Links

- [Maps Control](#)
- [iOS Maps](#)
- [Maps API](#)
- [Customized Pin \(sample\)](#)

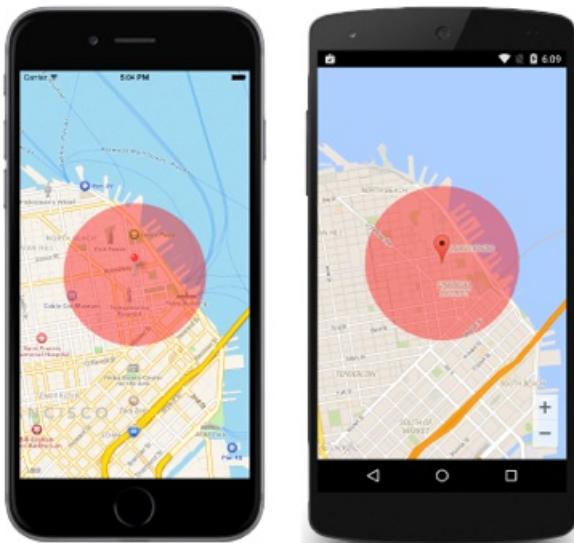
# Highlighting a Circular Area on a Map

7/12/2018 • 6 minutes to read • [Edit Online](#)

This article explains how to add a circular overlay to a map, to highlight a circular area of the map.

## Overview

An overlay is a layered graphic on a map. Overlays support drawing graphical content that scales with the map as it is zoomed. The following screenshots show the result of adding a circular overlay to a map:



When a `Map` control is rendered by a Xamarin.Forms application, in iOS the `MapRenderer` class is instantiated, which in turn instantiates a native `MKMapView` control. On the Android platform, the `MapRenderer` class instantiates a native `MapView` control. On the Universal Windows Platform (UWP), the `MapRenderer` class instantiates a native `MapControl`. The rendering process can be taken advantage of to implement platform-specific map customizations by creating a custom renderer for a `Map` on each platform. The process for doing this is as follows:

1. [Create](#) a Xamarin.Forms custom map.
2. [Consume](#) the custom map from Xamarin.Forms.
3. [Customize](#) the map by creating a custom renderer for the map on each platform.

### NOTE

`Xamarin.Forms.Maps` must be initialized and configured before use. For more information, see [Maps Control](#)

For information about customizing a map using a custom renderer, see [Customizing a Map Pin](#).

## Creating the Custom Map

Create a `CustomCircle` class that has `Position` and `Radius` properties:

```
public class CustomCircle
{
    public Position Position { get; set; }
    public double Radius { get; set; }
}
```

Then, create a subclass of the `Map` class, that adds a property of type `CustomCircle`:

```
public class CustomMap : Map
{
    public CustomCircle Circle { get; set; }
}
```

## Consuming the Custom Map

Consume the `CustomMap` control by declaring an instance of it in the XAML page instance:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
    x:Class="MapOverlay.MapPath">
<ContentPage.Content>
    <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
        HeightRequest="{x:Static local:App.ScreenHeight}" />
</ContentPage.Content>
</ContentPage>
```

Alternatively, consume the `CustomMap` control by declaring an instance of it in the C# page instance:

```
public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}
```

Initialize the `CustomMap` control as required:

```
public partial class MapPage : ContentPage
{
    public MapPage ()
    {
        ...
        var pin = new Pin {
            Type = PinType.Place,
            Position = new Position (37.79752, -122.40183),
            Label = "Xamarin San Francisco Office",
            Address = "394 Pacific Ave, San Francisco CA"
        };

        var position = new Position (37.79752, -122.40183);
        customMap.Circle = new CustomCircle {
            Position = position,
            Radius = 1000
        };

        customMap.Pins.Add (pin);
        customMap.MoveToRegion (MapSpan.FromCenterAndRadius (position, Distance.FromMiles (1.0)));
    }
}
```

This initialization adds `Pin` and `CustomCircle` instances to the custom map, and positions the map's view with the `MoveToRegion` method, which changes the position and zoom level of the map by creating a `MapSpan` from a `Position` and a `Distance`.

## Customizing the Map

A custom renderer must now be added to each application project to add the circular overlay to the map.

### Creating the Custom Renderer on iOS

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the circular overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKCircleRenderer circleRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    circleRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                var circle = formsMap.Circle;

                nativeMap.OverlayRenderer = GetOverlayRenderer();

                var circleOverlay = MKCircle.Circle(new
CoreLocation.CLLocationCoordinate2D(circle.Position.Latitude, circle.Position.Longitude), circle.Radius);
                nativeMap.AddOverlay(circleOverlay);
            }
        }
        ...
    }
}
```

This method performs the following configuration, provided that the custom renderer is attached to a new Xamarin.Forms element:

- The `MKMapView.OverlayRenderer` property is set to a corresponding delegate.
- The circle is created by setting a static `MKCircle` object that specifies the center of the circle, and the radius of the circle in meters.
- The circle is added to the map by calling the `MKMapView.AddOverlay` method.

Then, implement the `GetOverlayRenderer` method to customize the rendering of the overlay:

```
public class CustomMapRenderer : MapRenderer
{
    MKCircleRenderer circleRenderer;
    ...

    MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
    {
        if (circleRenderer == null && !Equals(overlayWrapper, null)) {
            var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
            circleRenderer = new MKCircleRenderer(overlay as MKCircle) {
                FillColor = UIColor.Red,
                Alpha = 0.4f
            };
        }
        return circleRenderer;
    }
}
```

### Creating the Custom Renderer on Android

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` and `OnMapReady` methods to add the circular overlay:

```

[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        CustomCircle circle;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void
OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Xamarin.Forms.Maps.Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                circle = formsMap.Circle;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var circleOptions = new CircleOptions();
            circleOptions.InvokeCenter(new LatLng(circle.Position.Latitude, circle.Position.Longitude));
            circleOptions.InvokeRadius(circle.Radius);
            circleOptions.InvokeFillColor(0X66FF0000);
            circleOptions.InvokeStrokeColor(0X66FF0000);
            circleOptions.InvokeStrokeWidth(0);

            NativeMap.AddCircle(circleOptions);
        }
    }
}

```

The `OnElementChanged` method calls the `MapView.GetMapAsync` method, which gets the underlying `GoogleMap` that is tied to the view, provided that the custom renderer is attached to a new Xamarin.Forms element. Once the `GoogleMap` instance is available, the `onMapReady` method will be invoked, where the circle is created by instantiating a `CircleOptions` object that specifies the center of the circle, and the radius of the circle in meters. The circle is then added to the map by calling the `NativeMap.AddCircle` method.

#### **Creating the Custom Renderer on the Universal Windows Platform**

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the circular overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        const int EarthRadiusInMeters = 6371000;

        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }
            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;
                var circle = formsMap.Circle;

                var coordinates = new List<BasicGeoposition>();
                var positions = GenerateCircleCoordinates(circle.Position, circle.Radius);
                foreach (var position in positions)
                {
                    coordinates.Add(new BasicGeoposition { Latitude = position.Latitude, Longitude = position.Longitude });
                }

                var polygon = new MapPolygon();
                polygon.FillColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeThickness = 5;
                polygon.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polygon);
            }
        }
        // GenerateCircleCoordinates helper method (below)
    }
}
```

This method performs the following operations, provided that the custom renderer is attached to a new Xamarin.Forms element:

- The circle position and radius are retrieved from the `CustomMap.Circle` property and passed to the `GenerateCircleCoordinates` method, which generates latitude and longitude coordinates for the circle perimeter. The code for this helper method is shown below.
- The circle perimeter coordinates are converted into a `List` of `BasicGeoposition` coordinates.
- The circle is created by instantiating a `MapPolygon` object. The `MapPolygon` class is used to display a multi-point shape on the map by setting its `Path` property to a `Geopath` object that contains the shape coordinates.
- The polygon is rendered on the map by adding it to the `MapControl.MapElements` collection.

```

List<Position> GenerateCircleCoordinates(Position position, double radius)
{
    double latitude = position.Latitude.ToRadians();
    double longitude = position.Longitude.ToRadians();
    double distance = radius / EarthRadiusInMeters;
    var positions = new List<Position>();

    for (int angle = 0; angle <=360; angle++)
    {
        double angleInRadians = ((double)angle).ToRadians();
        double latitudeInRadians = Math.Asin(Math.Sin(latitude) * Math.Cos(distance) + Math.Cos(latitude) *
Math.Sin(distance) * Math.Cos(angleInRadians));
        double longitudeInRadians = longitude + Math.Atan2(Math.Sin(angleInRadians) * Math.Sin(distance) *
Math.Cos(latitude), Math.Cos(distance) - Math.Sin(latitude) * Math.Sin(latitudeInRadians));

        var pos = new Position(latitudeInRadians.ToDegrees(), longitudeInRadians.ToDegrees());
        positions.Add(pos);
    }

    return positions;
}

```

## Summary

This article explained how to add a circular overlay to a map, to highlight a circular area of the map.

## Related Links

- [Circular Map Overlay \(sample\)](#)
- [Customizing a Map Pin](#)
- [Xamarin.Forms.Maps](#)

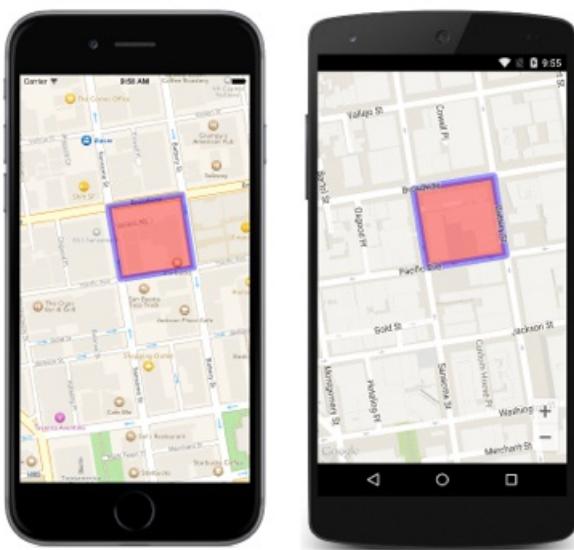
# Highlighting a Region on a Map

7/12/2018 • 6 minutes to read • [Edit Online](#)

This article explained how to add a polygon overlay to a map, to highlight a region on the map. Polygons are a closed shape and have their interiors filled in.

## Overview

An overlay is a layered graphic on a map. Overlays support drawing graphical content that scales with the map as it is zoomed. The following screenshots show the result of adding a polygon overlay to a map:



When a `Map` control is rendered by a Xamarin.Forms application, in iOS the `MapRenderer` class is instantiated, which in turn instantiates a native `MKMapView` control. On the Android platform, the `MapRenderer` class instantiates a native `MapView` control. On the Universal Windows Platform (UWP), the `MapRenderer` class instantiates a native `MapControl1`. The rendering process can be taken advantage of to implement platform-specific map customizations by creating a custom renderer for a `Map` on each platform. The process for doing this is as follows:

1. [Create](#) a Xamarin.Forms custom map.
2. [Consume](#) the custom map from Xamarin.Forms.
3. [Customize](#) the map by creating a custom renderer for the map on each platform.

### NOTE

`Xamarin.Forms.Maps` must be initialized and configured before use. For more information, see [Maps Control](#).

For information about customizing a map using a custom renderer, see [Customizing a Map Pin](#).

### Creating the Custom Map

Create a subclass of the `Map` class, that adds a `ShapeCoordinates` property:

```

public class CustomMap : Map
{
    public List<Position> ShapeCoordinates { get; set; }

    public CustomMap ()
    {
        ShapeCoordinates = new List<Position> ();
    }
}

```

The `ShapeCoordinates` property will store a collection of coordinates that define the region to be highlighted.

## Consuming the Custom Map

Consume the `CustomMap` control by declaring an instance of it in the XAML page instance:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
    x:Class="MapOverlay.MapPage">
    <ContentPage.Content>
        <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
            HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>

```

Alternatively, consume the `CustomMap` control by declaring an instance of it in the C# page instance:

```

public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = AppScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}

```

Initialize the `CustomMap` control as required:

```

public partial class MapPage : ContentPage
{
    public MapPage ()
    {
        ...
        customMap.ShapeCoordinates.Add (new Position (37.797513, -122.402058));
        customMap.ShapeCoordinates.Add (new Position (37.798433, -122.402256));
        customMap.ShapeCoordinates.Add (new Position (37.798582, -122.401071));
        customMap.ShapeCoordinates.Add (new Position (37.797658, -122.400888));

        customMap.MoveToRegion (MapSpan.FromCenterAndRadius (new Position (37.79752, -122.40183),
            Distance.FromMiles (0.1)));
    }
}

```

This initialization specifies a series of latitude and longitude coordinates, to define the region of the map to be highlighted. It then positions the map's view with the `MoveToRegion` method, which changes the position and zoom

level of the map by creating a `MapSpan` from a `Position` and a `Distance`.

## Customizing the Map

A custom renderer must now be added to each application project to add the polygon overlay to the map.

### Creating the Custom Renderer on iOS

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the polygon overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKPolygonRenderer polygonRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    polygonRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;

                nativeMap.OverlayRenderer = GetOverlayRenderer();

                CLLocationCoordinate2D[] coords = new CLLocationCoordinate2D[formsMap.ShapeCoordinates.Count];

                int index = 0;
                foreach (var position in formsMap.ShapeCoordinates)
                {
                    coords[index] = new CLLocationCoordinate2D(position.Latitude, position.Longitude);
                    index++;
                }

                var blockOverlay = MKPolygon.FromCoordinates(coords);
                nativeMap.AddOverlay(blockOverlay);
            }
        }
    ...
}
```

This method performs the following configuration, provided that the custom renderer is attached to a new Xamarin.Forms element:

- The `MKMapView.OverlayRenderer` property is set to a corresponding delegate.
- The collection of latitude and longitude coordinates are retrieved from the `CustomMap.ShapeCoordinates` property and stored as an array of `CLLocationCoordinate2D` instances.
- The polygon is created by calling the static `MKPolygon.FromCoordinates` method, which specifies the latitude and longitude of each point.
- The polygon is added to the map by calling the `MKMapView.AddOverlay` method. This method automatically closes the polygon by drawing a line that connects the first and last points.

Then, implement the `GetOverlayRenderer` method to customize the rendering of the overlay:

```
public class CustomMapRenderer : MapRenderer
{
    MKPolygonRenderer polygonRenderer;
    ...

    MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
    {
        if (polygonRenderer == null && !Equals(overlayWrapper, null)) {
            var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
            polygonRenderer = new MKPolygonRenderer(overlay as MKPolygon) {
                FillColor = UIColor.Red,
                StrokeColor = UIColor.Blue,
                Alpha = 0.4f,
                LineWidth = 9
            };
        }
        return polygonRenderer;
    }
}
```

#### Creating the Custom Renderer on Android

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` and `OnMapReady` methods to add the polygon overlay:

```

[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        List<Position> shapeCoordinates;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                shapeCoordinates = formsMap.ShapeCoordinates;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var polygonOptions = new PolygonOptions();
            polygonOptions.InvokeFillColor(0x66FF0000);
            polygonOptions.InvokeStrokeColor(0x660000FF);
            polygonOptions.InvokeStrokeWidth(30.0f);

            foreach (var position in shapeCoordinates)
            {
                polygonOptions.Add(new LatLng(position.Latitude, position.Longitude));
            }
            NativeMap.AddPolygon(polygonOptions);
        }
    }
}

```

The `OnElementChanged` method retrieves the collection of latitude and longitude coordinates from the `CustomMap.ShapeCoordinates` property and stores them in a member variable. It then calls the `MapView.GetMapAsync` method, which gets the underlying `GoogleMap` that is tied to the view, provided that the custom renderer is attached to a new Xamarin.Forms element. Once the `GoogleMap` instance is available, the `OnMapReady` method will be invoked, where the polygon is created by instantiating a `PolygonOptions` object that specifies the latitude and longitude of each point. The polygon is then added to the map by calling the `NativeMap.AddPolygon` method. This method automatically closes the polygon by drawing a line that connects the first and last points.

#### **Creating the Custom Renderer on the Universal Windows Platform**

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the polygon overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;

                var coordinates = new List<BasicGeoposition>();
                foreach (var position in formsMap.ShapeCoordinates)
                {
                    coordinates.Add(new BasicGeoposition() { Latitude = position.Latitude, Longitude = position.Longitude });
                }

                var polygon = new MapPolygon();
                polygon.FillColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeColor = Windows.UI.Color.FromArgb(128, 0, 0, 255);
                polygon.StrokeThickness = 5;
                polygon.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polygon);
            }
        }
    }
}
```

This method performs the following operations, provided that the custom renderer is attached to a new `Xamarin.Forms` element:

- The collection of latitude and longitude coordinates are retrieved from the `CustomMap.ShapeCoordinates` property and converted into a `List` of `BasicGeoposition` coordinates.
- The polygon is created by instantiating a `MapPolygon` object. The `MapPolygon` class is used to display a multi-point shape on the map by setting its `Path` property to a `Geopath` object that contains the shape coordinates.
- The polygon is rendered on the map by adding it to the `MapControl.MapElements` collection. Note that the polygon will be automatically closed by drawing a line that connects the first and last points.

## Summary

This article explained how to add a polygon overlay to a map, to highlight a region of the map. Polygons are a closed shape and have their interiors filled in.

## Related Links

- [Polygon Map Overlay \(sample\)](#)
- [Customizing a Map Pin](#)
- [Xamarin.Forms.Maps](#)

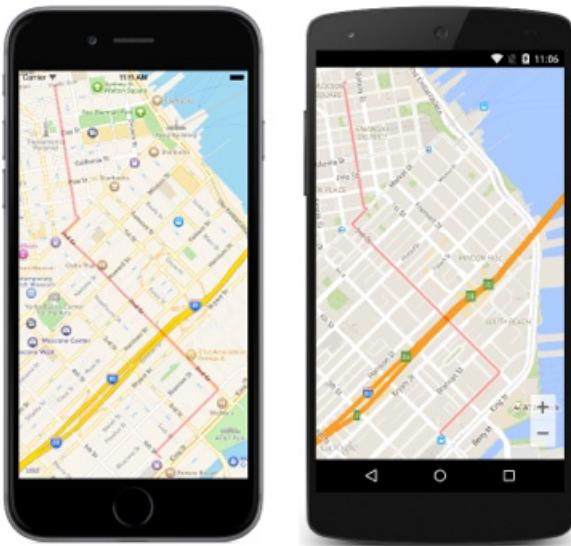
# Highlighting a Route on a Map

7/12/2018 • 5 minutes to read • [Edit Online](#)

This article explains how to add a polyline overlay to a map. A polyline overlay is a series of connected line segments that are typically used to show a route on a map, or form any shape that's required.

## Overview

An overlay is a layered graphic on a map. Overlays support drawing graphical content that scales with the map as it is zoomed. The following screenshots show the result of adding a polyline overlay to a map:



When a `Map` control is rendered by a Xamarin.Forms application, in iOS the `MapRenderer` class is instantiated, which in turn instantiates a native `MKMapView` control. On the Android platform, the `MapRenderer` class instantiates a native `MapView` control. On the Universal Windows Platform (UWP), the `MapRenderer` class instantiates a native `MapControl1`. The rendering process can be taken advantage of to implement platform-specific map customizations by creating a custom renderer for a `Map` on each platform. The process for doing this is as follows:

1. [Create](#) a Xamarin.Forms custom map.
2. [Consume](#) the custom map from Xamarin.Forms.
3. [Customize](#) the map by creating a custom renderer for the map on each platform.

### NOTE

`Xamarin.Forms.Maps` must be initialized and configured before use. For more information, see [Maps Control](#).

For information about customizing a map using a custom renderer, see [Customizing a Map Pin](#).

### Creating the Custom Map

Create a subclass of the `Map` class, that adds a `RouteCoordinates` property:

```

public class CustomMap : Map
{
    public List<Position> RouteCoordinates { get; set; }

    public CustomMap ()
    {
        RouteCoordinates = new List<Position> ();
    }
}

```

The `RouteCoordinates` property will store a collection of coordinates that define the route to be highlighted.

## Consuming the Custom Map

Consume the `CustomMap` control by declaring an instance of it in the XAML page instance:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
    x:Class="MapOverlay.MapPage">
    <ContentPage.Content>
        <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
            HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>

```

Alternatively, consume the `CustomMap` control by declaring an instance of it in the C# page instance:

```

public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}

```

Initialize the `CustomMap` control as required:

```

public partial class MapPage : ContentPage
{
    public MapPage ()
    {
        ...
        customMap.RouteCoordinates.Add (new Position (37.785559, -122.396728));
        customMap.RouteCoordinates.Add (new Position (37.780624, -122.390541));
        customMap.RouteCoordinates.Add (new Position (37.777113, -122.394983));
        customMap.RouteCoordinates.Add (new Position (37.776831, -122.394627));

        customMap.MoveToRegion (MapSpan.FromCenterAndRadius (new Position (37.79752, -122.40183),
            Distance.FromMiles (1.0)));
    }
}

```

This initialization specifies a series of latitude and longitude coordinates, to define the route on the map to be highlighted. It then positions the map's view with the `MoveToRegion` method, which changes the position and zoom

level of the map by creating a `MapSpan` from a `Position` and a `Distance`.

## Customizing the Map

A custom renderer must now be added to each application project to add the polyline overlay to the map.

### Creating the Custom Renderer on iOS

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the polyline overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKPolylineRenderer polylineRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    polylineRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                nativeMap.OverlayRenderer = GetOverlayRenderer();

                CLLocationCoordinate2D[] coords = new CLLocationCoordinate2D[formsMap.RouteCoordinates.Count];
                int index = 0;
                foreach (var position in formsMap.RouteCoordinates)
                {
                    coords[index] = new CLLocationCoordinate2D(position.Latitude, position.Longitude);
                    index++;
                }

                var routeOverlay = MKPolyline.FromCoordinates(coords);
                nativeMap.AddOverlay(routeOverlay);
            }
        }
        ...
    }
}
```

This method performs the following configuration, provided that the custom renderer is attached to a new Xamarin.Forms element:

- The `MKMapView.OverlayRenderer` property is set to a corresponding delegate.
- The collection of latitude and longitude coordinates are retrieved from the `CustomMap.RouteCoordinates` property and stored as an array of `CLLocationCoordinate2D` instances.
- The polyline is created by calling the static `MKPolyline.FromCoordinates` method, which specifies the latitude and longitude of each point.
- The polyline is added to the map by calling the `MKMapView.AddOverlay` method.

Then, implement the `GetOverlayRenderer` method to customize the rendering of the overlay:

```
public class CustomMapRenderer : MapRenderer
{
    MKPolylineRenderer polylineRenderer;
    ...

    MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
    {
        if (polylineRenderer == null && !Equals(overlayWrapper, null)) {
            var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
            polylineRenderer = new MKPolylineRenderer(overlay as MKPolyline) {
                FillColor = UIColor.Blue,
                StrokeColor = UIColor.Red,
                LineWidth = 3,
                Alpha = 0.4f
            };
        }
        return polylineRenderer;
    }
}
```

#### Creating the Custom Renderer on Android

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` and `OnMapReady` methods to add the polyline overlay:

```

[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        List<Position> routeCoordinates;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                routeCoordinates = formsMap.RouteCoordinates;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var polylineOptions = new PolylineOptions();
            polylineOptions.InvokeColor(0x66FF0000);

            foreach (var position in routeCoordinates)
            {
                polylineOptions.Add(new LatLng(position.Latitude, position.Longitude));
            }

            NativeMap.AddPolyline(polylineOptions);
        }
    }
}

```

The `OnElementChanged` method retrieves the collection of latitude and longitude coordinates from the `CustomMap.RouteCoordinates` property and stores them in a member variable. It then calls the `MapView.GetMapAsync` method, which gets the underlying `GoogleMap` that is tied to the view, provided that the custom renderer is attached to a new Xamarin.Forms element. Once the `GoogleMap` instance is available, the `OnMapReady` method will be invoked, where the polyline is created by instantiating a `PolylineOptions` object that specifies the latitude and longitude of each point. The polyline is then added to the map by calling the `NativeMap.AddPolyline` method.

#### **Creating the Custom Renderer on the Universal Windows Platform**

Create a subclass of the `MapRenderer` class and override its `OnElementChanged` method to add the polyline overlay:

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;

                var coordinates = new List<BasicGeoposition>();
                foreach (var position in formsMap.RouteCoordinates)
                {
                    coordinates.Add(new BasicGeoposition() { Latitude = position.Latitude, Longitude = position.Longitude });
                }

                var polyline = new MapPolyline();
                polyline.StrokeColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polyline.StrokeThickness = 5;
                polyline.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polyline);
            }
        }
    }
}
```

This method performs the following operations, provided that the custom renderer is attached to a new `Xamarin.Forms` element:

- The collection of latitude and longitude coordinates are retrieved from the `CustomMap.RouteCoordinates` property and converted into a `List` of `BasicGeoposition` coordinates.
- The polyline is created by instantiating a `MapPolyline` object. The `MapPolygon` class is used to display a line on the map by setting its `Path` property to a `Geopath` object that contains the line coordinates.
- The polyline is rendered on the map by adding it to the `MapControl.MapElements` collection.

## Summary

This article explained how to add a polyline overlay to a map, to show a route on a map, or form any shape that's required.

## Related Links

- [Polyline Map Overlay \(sample\)](#)
- [Customizing a Map Pin](#)
- [Xamarin.Forms.Maps](#)

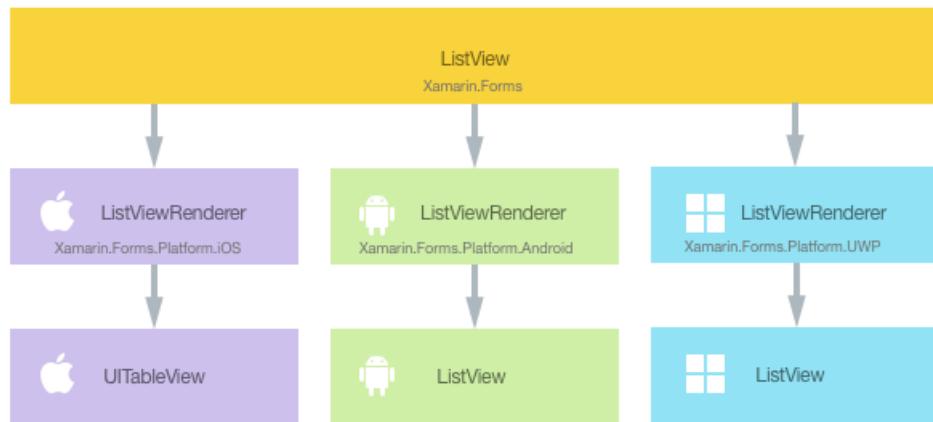
# Customizing a ListView

7/12/2018 • 16 minutes to read • [Edit Online](#)

A `Xamarin.Forms.ListView` is a view that displays a collection of data as a vertical list. This article demonstrates how to create a custom renderer that encapsulates platform-specific list controls and native cell layouts, allowing more control over native list control performance.

Every `Xamarin.Forms` view has an accompanying renderer for each platform that creates an instance of a native control. When a `ListView` is rendered by a `Xamarin.Forms` application, in iOS the `ListViewRenderer` class is instantiated, which in turn instantiates a native `UITableView` control. On the Android platform, the `ListViewRenderer` class instantiates a native `ListView` control. On the Universal Windows Platform (UWP), the `ListViewRenderer` class instantiates a native `ListView` control. For more information about the renderer and native control classes that `Xamarin.Forms` controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `ListView` control and the corresponding native controls that implement it:



The rendering process can be taken advantage of to implement platform-specific customizations by creating a custom renderer for a `ListView` on each platform. The process for doing this is as follows:

1. [Create](#) a `Xamarin.Forms` custom control.
2. [Consume](#) the custom control from `Xamarin.Forms`.
3. [Create](#) the custom renderer for the control on each platform.

Each item will now be discussed in turn, to implement a `NativeListView` renderer that takes advantage of platform-specific list controls and native cell layouts. This scenario is useful when porting an existing native app that contains list and cell code that can be re-used. In addition, it allows detailed customization of list control features that can affect performance, such as data virtualization.

## Creating the Custom ListView Control

A custom `ListView` control can be created by subclassing the `ListView` class, as shown in the following code example:

```

public class NativeListView : ListView
{
    public static readonly BindableProperty ItemsProperty =
        BindableProperty.Create ("Items", typeof(IEnumerable<DataSource>), typeof(NativeListView), new
List<DataSource> ());

    public IEnumerable<DataSource> Items {
        get { return (IEnumerable<DataSource>)GetValue (ItemsProperty); }
        set { SetValue (ItemsProperty, value); }
    }

    public event EventHandler<SelectedItemChangedEventArgs> ItemSelected;

    public void NotifyItemSelected (object item)
    {
        if (ItemSelected != null) {
            ItemSelected (this, new SelectedItemChangedEventArgs (item));
        }
    }
}

```

The `NativeListView` is created in the .NET Standard library project and defines the API for the custom control. This control exposes an `Items` property that is used for populating the `ListView` with data, and which can be data bound to for display purposes. It also exposes an `ItemSelected` event that will be fired whenever an item is selected in a platform-specific native list control. For more information about data binding, see [Data Binding Basics](#).

## Consuming the Custom Control

The `NativeListView` custom control can be referenced in XAML in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the control. The following code example shows how the `NativeListView` custom control can be consumed by a XAML page:

```

<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <ContentPage.Content>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Label Text="{x:Static local:App.Description}" HorizontalTextAlignment="Center" />
            <local:NativeListView Grid.Row="1" x:Name="nativeListView" ItemSelected="OnItemSelected"
VerticalOptions="FillAndExpand" />
        </Grid>
    </ContentPage.Content>
</ContentPage>

```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom control.

The following code example shows how the `NativeListView` custom control can be consumed by a C# page:

```

public class MainPageCS : ContentPage
{
    NativeListView nativeListView;

    public MainPageCS()
    {
        nativeListView = new NativeListView
        {
            Items = DataSource.GetList(),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
                Padding = new Thickness(0, 20, 0, 0);
                break;
            case Device.Android:
            case Device.UWP:
                Padding = new Thickness(0);
                break;
        }

        Content = new Grid
        {
            RowDefinitions = {
                new RowDefinition { Height = GridLength.Auto },
                new RowDefinition { Height = new GridLength (1, GridUnitType.Star) }
            },
            Children = {
                new Label { Text = App.Description, HorizontalTextAlignment = TextAlignment.Center },
                nativeListView
            }
        };
        nativeListView.ItemSelected += OnItemSelected;
    }
    ...
}

```

The `NativeListView` custom control uses platform-specific custom renderers to display a list of data, which is populated through the `Items` property. Each row in the list contains three items of data – a name, a category, and an image filename. The layout of each row in the list is defined by the platform-specific custom renderer.

#### **NOTE**

Because the `NativeListView` custom control will be rendered using platform-specific list controls that include scrolling ability, the custom control should not be hosted in scrollable layout controls such as the `ScrollView`.

A custom renderer can now be added to each application project to create platform-specific list controls and native cell layouts.

## Creating the Custom Renderer on each Platform

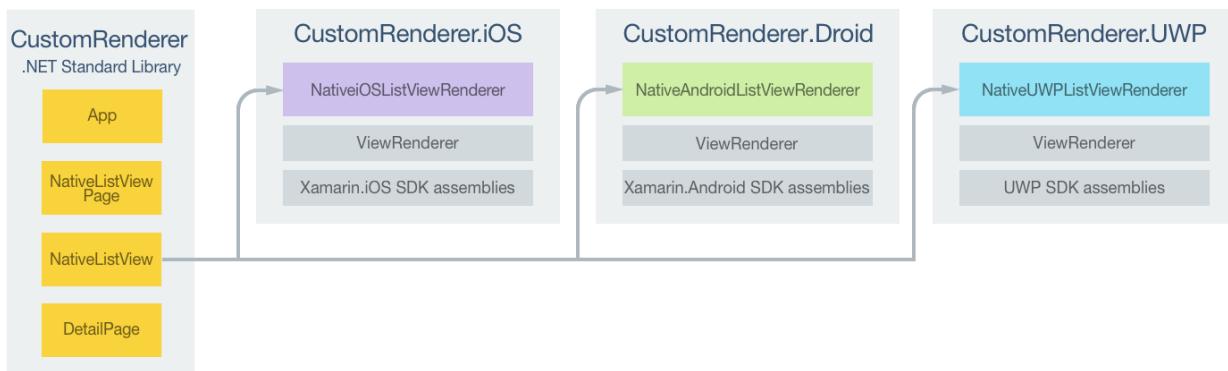
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `ListViewRenderer` class that renders the custom control.
2. Override the `OnElementChanged` method that renders the custom control and write logic to customize it. This method is called when the corresponding `Xamarin.Forms.ListView` is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the `Xamarin.Forms` custom control. This attribute is used to register the custom renderer with `Xamarin.Forms`.

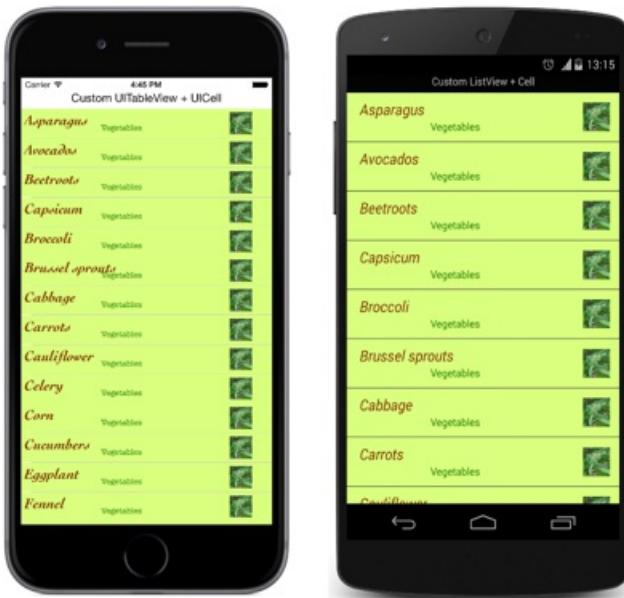
## NOTE

It is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the cell's base class will be used.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `NativeListView` custom control is rendered by platform-specific renderer classes, which all derive from the `ListViewRenderer` class for each platform. This results in each `NativeListView` custom control being rendered with platform-specific list controls and native cell layouts, as shown in the following screenshots:



The `ListViewRenderer` class exposes the `OnElementChanged` method, which is called when the Xamarin.Forms custom control is created to render the corresponding native control. This method takes an `ElementChangedEventArgs` parameter, that contains `OldElement` and `NewElement` properties. These properties represent the Xamarin.Forms element that the renderer was attached to, and the Xamarin.Forms element that the renderer is attached to, respectively. In the sample application, the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `NativeListView` instance.

An overridden version of the `OnElementChanged` method, in each platform-specific renderer class, is the place to perform the native control customization. A typed reference to the native control being used on the platform can be accessed through the `control` property. In addition, a reference to the Xamarin.Forms control that's being rendered can be obtained through the `Element` property.

Care must be taken when subscribing to event handlers in the `OnElementChanged` method, as demonstrated in the following code example:

```

protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
{
    base.OnElementChanged (e);

    if (e.OldElement != null) {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null) {
        // Configure the native control and subscribe to event handlers
    }
}

```

The native control should only be configured and event handlers subscribed to when the custom renderer is attached to a new Xamarin.Forms element. Similarly, any event handlers that were subscribed to should be unsubscribed from only when the element the renderer is attached to changes. Adopting this approach will help to create a custom renderer that doesn't suffer from memory leaks.

An overridden version of the `OnElementPropertyChanged` method, in each platform-specific renderer class, is the place to respond to bindable property changes on the Xamarin.Forms custom control. A check for the property that's changed should always be made, as this override can be called many times.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms custom control being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific custom renderer class.

### Creating the Custom Renderer on iOS

The following code example shows the custom renderer for the iOS platform:

```

[assembly: ExportRenderer (typeof(NativeListView), typeof(NativeiOSListViewRenderer))]
namespace CustomRenderer.iOS
{
    public class NativeiOSListViewRenderer : ListViewRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged (e);

            if (e.OldElement != null) {
                // Unsubscribe
            }

            if (e.NewElement != null) {
                Control.Source = new NativeiOSListViewSource (e.NewElement as NativeListView);
            }
        }
    }
}

```

The `UITableView` control is configured by creating an instance of the `NativeiOSListViewSource` class, provided that the custom renderer is attached to a new Xamarin.Forms element. This class provides data to the `UITableView` control by overriding the `RowsInSection` and `GetCell` methods from the `UITableViewSource` class, and by exposing an `Items` property that contains the list of data to be displayed. The class also provides a `RowSelected` method override that invokes the `ItemSelected` event provided by the `NativeListView` custom control. For more information about the method overrides, see [Subclassing UITableViewSource](#). The `GetCell` method returns a `UITableViewCellView` that's populated with data for each row in the list, and is shown in the following code example:

```

public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
{
    // request a recycled cell to save memory
    NativeiOSListTableViewCell cell = tableView.DequeueReusableCell (cellIdentifier) as NativeiOSListTableViewCell;

    // if there are no cells to reuse, create a new one
    if (cell == null) {
        cell = new NativeiOSListTableViewCell (cellIdentifier);
    }

    if (String.IsNullOrEmpty (tableItems [indexPath.Row].ImageFilename)) {
        cell.UpdateCell (tableItems [indexPath.Row].Name
            , tableItems [indexPath.Row].Category
            , null);
    } else {
        cell.UpdateCell (tableItems [indexPath.Row].Name
            , tableItems [indexPath.Row].Category
            , UIImage.FromFile ("Images/" + tableItems [indexPath.Row].ImageFilename + ".jpg"));
    }

    return cell;
}

```

This method creates a `NativeiOSListTableViewCell` instance for each row of data that will be displayed on the screen. The `NativeiOSCell` instance defines the layout of each cell and the cell's data. When a cell disappears from the screen due to scrolling, the cell will be made available for reuse. This avoids wasting memory by ensuring that there are only `NativeiOSCell` instances for the data being displayed on the screen, rather than all of the data in the list. For more information about cell reuse, see [Cell Reuse](#). The `GetCell` method also reads the `ImageFilename` property of each row of data, provided that it exists, and reads the image and stores it as a `UIImage` instance, before updating the `NativeiOSListTableViewCell` instance with the data (name, category, and image) for the row.

The `NativeiOSListTableViewCell` class defines the layout for each cell, and is shown in the following code example:

```

public class NativeiOSListTableViewCell : UITableViewCell
{
    UILabel headingLabel, subheadingLabel;
    UIImageView imageView;

    public NativeiOSListTableViewCell (NSString cellId) : base (UITableViewCellStyle.Default, cellId)
    {
        SelectionStyle = UITableViewCellSelectionStyle.Gray;

        ContentView.BackgroundColor = UIColor.FromRGB (218, 255, 127);

        imageView = new UIImageView ();

        headingLabel = new UILabel () {
            Font = UIFont.FromName ("Cochin-BoldItalic", 22f),
            TextColor = UIColor.FromRGB (127, 51, 0),
            BackgroundColor = UIColor.Clear
        };

        subheadingLabel = new UILabel () {
            Font = UIFont.FromName ("AmericanTypewriter", 12f),
            TextColor = UIColor.FromRGB (38, 127, 0),
            TextAlignment = UITextAlignment.Center,
            BackgroundColor = UIColor.Clear
        };

        ContentView.Add (headingLabel);
        ContentView.Add (subheadingLabel);
        ContentView.Add (imageView);
    }

    public void UpdateCell (string caption, string subtitle, UIImage image)
    {
        headingLabel.Text = caption;
        subheadingLabel.Text = subtitle;
        imageView.Image = image;
    }

    public override void LayoutSubviews ()
    {
        base.LayoutSubviews ();

        headingLabel.Frame = new CoreGraphics.CGRect (5, 4, ContentView.Bounds.Width - 63, 25);
        subheadingLabel.Frame = new CoreGraphics.CGRect (100, 18, 100, 20);
        imageView.Frame = new CoreGraphics.CGRect (ContentView.Bounds.Width - 63, 5, 33, 33);
    }
}

```

This class defines the controls used to render the cell's contents, and their layout. The `NativeiOSListTableViewCell` constructor creates instances of `UILabel` and `UIImageView` controls, and initializes their appearance. These controls are used to display each row's data, with the `UpdateCell` method being used to set this data on the `UILabel` and `UIImageView` instances. The location of these instances is set by the overridden `LayoutSubviews` method, by specifying their coordinates within the cell.

#### **Responding to a Property Change on the Custom Control**

If the `NativeListView.Items` property changes, due to items being added to or removed from the list, the custom renderer needs to respond by displaying the changes. This can be accomplished by overriding the `OnElementPropertyChanged` method, which is shown in the following code example:

```

protected override void OnElementPropertyChanged (object sender,
System.ComponentModel.PropertyChangedEventArgs e)
{
    base.OnElementPropertyChanged (sender, e);

    if (e.PropertyName == NativeListView.ItemsProperty.PropertyName) {
        Control.Source = new NativeiOSListViewSource (Element as NativeListView);
    }
}

```

The method creates a new instance of the `NativeiOSListViewSource` class that provides data to the `UITableView` control, provided that the bindable `NativeListView.Items` property has changed.

### Creating the Custom Renderer on Android

The following code example shows the custom renderer for the Android platform:

```

[assembly: ExportRenderer(typeof(NativeListView), typeof(NativeAndroidListViewRenderer))]
namespace CustomRenderer.Droid
{
    public class NativeAndroidListViewRenderer : ListViewRenderer
    {
        Context _context;

        public NativeAndroidListViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // unsubscribe
                Control.ItemClick -= OnItemClick;
            }

            if (e.NewElement != null)
            {
                // subscribe
                Control.Adapter = new NativeAndroidListViewAdapter(_context as Android.App.Activity,
e.NewElement as NativeListView);
                Control.ItemClick += OnItemClick;
            }
            ...
        }

        void OnItemClick(object sender, Android.Widget.AdapterView.ItemClickEventArgs e)
        {
            ((NativeListView)Element).NotifyItemSelected(((NativeListView)Element).Items.ToList()[e.Position - 1]);
        }
    }
}

```

The native `ListView` control is configured provided that the custom renderer is attached to a new `Xamarin.Forms` element. This configuration involves creating an instance of the `NativeAndroidListViewAdapter` class that provides data to the native `ListView` control, and registering an event handler to process the `ItemClick` event. In turn, this handler will invoke the `ItemSelected` event provided by the `NativeListView` custom control. The `ItemClick` event is unsubscribed from if the `Xamarin.Forms` element the renderer is attached to changes.

The `NativeAndroidListViewAdapter` derives from the `BaseAdapter` class and exposes an `Items` property that contains the list of data to be displayed, as well as overriding the `Count`, `GetView`, `GetItemId`, and `this[int]` methods. For more information about these method overrides, see [Implementing a ListAdapter](#). The `GetView` method returns a view for each row, populated with data, and is shown in the following code example:

```
public override View GetView (int position, View convertView, ViewGroup parent)
{
    var item = tableItems [position];

    var view = convertView;
    if (view == null) {
        // no view to re-use, create new
        view = context.LayoutInflater.Inflate (Resource.Layout.NativeAndroidListViewCell, null);
    }
    view.FindViewById<TextView> (Resource.Id.Text1).Text = item.Name;
    view.FindViewById<TextView> (Resource.Id.Text2).Text = item.Category;

    // grab the old image and dispose of it
    if (view.FindViewById<ImageView> (Resource.Id.Image).Drawable != null) {
        using (var image = view.FindViewById<ImageView> (Resource.Id.Image).Drawable as BitmapDrawable) {
            if (image != null) {
                if (image.Bitmap != null) {
                    //image.Bitmap.Recycle ();
                    image.Bitmap.Dispose ();
                }
            }
        }
    }

    // If a new image is required, display it
    if (!String.IsNullOrWhiteSpace (item.ImageFilename)) {
        context.Resources.GetBitmapAsync (item.ImageFilename).ContinueWith ((t) => {
            var bitmap = t.Result;
            if (bitmap != null) {
                view.FindViewById<ImageView> (Resource.Id.Image).SetImageBitmap (bitmap);
                bitmap.Dispose ();
            }
        }, TaskScheduler.FromCurrentSynchronizationContext ());
    } else {
        // clear the image
        view.FindViewById<ImageView> (Resource.Id.Image).SetImageBitmap (null);
    }
}

return view;
}
```

The `GetView` method is called to return the cell to be rendered, as a `View`, for each row of data in the list. It creates a `View` instance for each row of data that will be displayed on the screen, with the appearance of the `View` instance being defined in a layout file. When a cell disappears from the screen due to scrolling, the cell will be made available for reuse. This avoids wasting memory by ensuring that there are only `View` instances for the data being displayed on the screen, rather than all of the data in the list. For more information about view reuse, see [Row View Re-use](#).

The `GetView` method also populates the `View` instance with data, including reading the image data from the filename specified in the `ImageFilename` property.

The layout of each cell displayed by the native `ListView` is defined in the `NativeAndroidListViewCell.axml` layout file, which is inflated by the `LayoutInflater.Inflate` method. The following code example shows the layout definition:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:background="@drawable/CustomSelector">
    <LinearLayout
        android:id="@+id/Text"
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="10dp">
        <TextView
            android:id="@+id/Text1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dp"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/Text2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="14dp"
            android:textColor="#FF267F00"
            android:paddingLeft="100dp" />
    </LinearLayout>
    <ImageView
        android:id="@+id/Image"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:padding="5dp"
        android:src="@drawable/icon"
        android:layout_alignParentRight="true" />
</RelativeLayout>

```

This layout specifies that two `TextView` controls and an `ImageView` control are used to display the cell's content. The two `TextView` controls are vertically oriented within a `LinearLayout` control, with all the controls being contained within a `RelativeLayout`.

#### **Responding to a Property Change on the Custom Control**

If the `NativeListView.Items` property changes, due to items being added to or removed from the list, the custom renderer needs to respond by displaying the changes. This can be accomplished by overriding the `OnElementPropertyChanged` method, which is shown in the following code example:

```

protected override void OnElementPropertyChanged (object sender,
System.ComponentModel.PropertyChangedEventArgs e)
{
    base.OnElementPropertyChanged (sender, e);

    if (e.PropertyName == NativeListView.ItemsProperty.PropertyName) {
        Control.Adapter = new NativeAndroidListViewAdapter (_context as Android.App.Activity, Element as
NativeListView);
    }
}

```

The method creates a new instance of the `NativeAndroidListViewAdapter` class that provides data to the native `ListView` control, provided that the bindable `NativeListView.Items` property has changed.

#### **Creating the Custom Renderer on UWP**

The following code example shows the custom renderer for UWP:

```

[assembly: ExportRenderer(typeof(NativeListView), typeof(NativeUWPListViewRenderer))]
namespace CustomRenderer.UWP
{
    public class NativeUWPListViewRenderer : ListViewRenderer
    {
        ListView listView;

        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged(e);

            listView = Control as ListView;

            if (e.OldElement != null)
            {
                // Unsubscribe
                listView.SelectionChanged -= OnSelectedItemChanged;
            }

            if (e.NewElement != null)
            {
                listViewSelectionMode = ListViewSelectionMode.Single;
                listView.IsEnabled = false;
                listView.ItemsSource = ((NativeListView)e.NewElement).Items;
                listView.ItemTemplate = App.Current.Resources["ListViewItemTemplate"] as
Windows.UI.Xaml.DataTemplate;
                // Subscribe
                listView.SelectionChanged += OnSelectedItemChanged;
            }
        }

        void OnSelectedItemChanged(object sender, SelectionChangedEventArgs e)
        {
            ((NativeListView)Element).NotifyItemSelected(listView.SelectedItem);
        }
    }
}

```

The native `ListView` control is configured provided that the custom renderer is attached to a new `Xamarin.Forms` element. This configuration involves setting how the native `ListView` control will respond to items being selected, populating the data displayed by the control, defining the appearance and contents of each cell, and registering an event handler to process the `SelectionChanged` event. In turn, this handler will invoke the `ItemSelected` event provided by the `NativeListView` custom control. The `SelectionChanged` event is unsubscribed from if the `Xamarin.Forms` element the renderer is attached to changes.

The appearance and contents of each native `ListView` cell are defined by a `DataTemplate` named `ListViewItemTemplate`. This `DataTemplate` is stored in the application-level resource dictionary, and is shown in the following code example:

```

<DataTemplate x:Key="ListViewItemTemplate">
    <Grid Background="#DAFF7F">
        <Grid.Resources>
            <local:ConcatImageExtensionConverter x:Name="ConcatImageExtensionConverter" />
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.20*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.ColumnSpan="2" Foreground="#7F3300" FontStyle="Italic" FontSize="22"
            VerticalAlignment="Top" Text="{Binding Name}" />
        <TextBlock Grid.RowSpan="2" Grid.Column="1" Foreground="#267F00" FontWeight="Bold" FontSize="12"
            VerticalAlignment="Bottom" Text="{Binding Category}" />
        <Image Grid.RowSpan="2" Grid.Column="2" HorizontalAlignment="Left" VerticalAlignment="Center" Source=
            "{Binding ImageFilename, Converter={StaticResource ConcatImageExtensionConverter}}" Width="50" Height="50" />
        <Line Grid.Row="1" Grid.ColumnSpan="3" X1="0" X2="1" Margin="30,20,0,0" StrokeThickness="1"
            Stroke="LightGray" Stretch="Fill" VerticalAlignment="Bottom" />
    </Grid>
</DataTemplate>

```

The `DataTemplate` specifies the controls used to display the contents of the cell, and their layout and appearance. Two `TextBlock` controls and an `Image` control are used to display the cell's content through data binding. In addition, an instance of the `ConcatImageExtensionConverter` is used to concatenate the `.jpg` file extension to each image file name. This ensures that the `Image` control can load and render the image when its `Source` property is set.

#### Responding to a Property Change on the Custom Control

If the `NativeListView.Items` property changes, due to items being added to or removed from the list, the custom renderer needs to respond by displaying the changes. This can be accomplished by overriding the `OnElementPropertyChanged` method, which is shown in the following code example:

```

protected override void OnElementPropertyChanged(object sender, System.ComponentModel.PropertyChangedEventArgs e)
{
    base.OnElementPropertyChanged(sender, e);

    if (e.PropertyName == NativeListView.ItemsProperty.PropertyName)
    {
        listView.ItemsSource = ((NativeListView)Element).Items;
    }
}

```

The method re-populates the native `ListView` control with the changed data, provided that the bindable `NativeListView.Items` property has changed.

## Summary

This article has demonstrated how to create a custom renderer that encapsulates platform-specific list controls and native cell layouts, allowing more control over native list control performance.

## Related Links

- [CustomRendererListView \(sample\)](#)

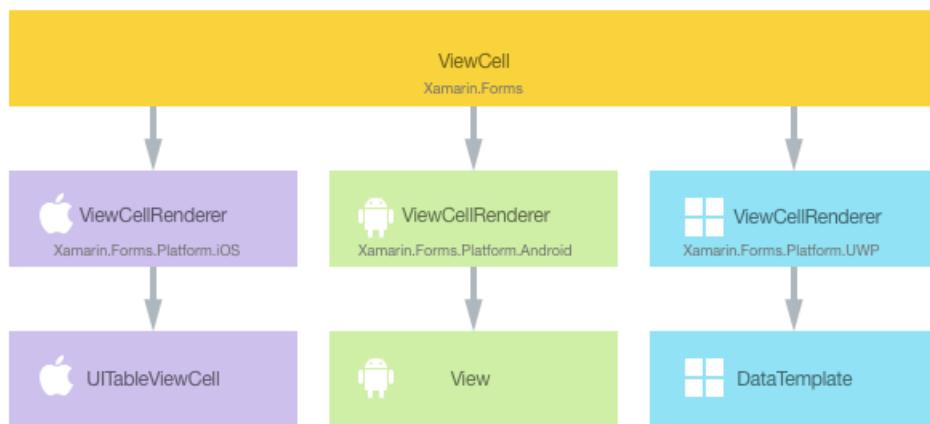
# Customizing a ViewCell

7/12/2018 • 14 minutes to read • [Edit Online](#)

A `Xamarin.Forms ViewCell` is a cell that can be added to a `ListView` or `TableView`, which contains a developer-defined view. This article demonstrates how to create a custom renderer for a `ViewCell` that's hosted inside a `Xamarin.Forms ListView` control. This stops the `Xamarin.Forms` layout calculations from being repeatedly called during `ListView` scrolling.

Every `Xamarin.Forms` cell has an accompanying renderer for each platform that creates an instance of a native control. When a `ViewCell` is rendered by a `Xamarin.Forms` application, in iOS the `ViewCellRenderer` class is instantiated, which in turn instantiates a native `UITableViewCell` control. On the Android platform, the `ViewCellRenderer` class instantiates a native `View` control. On the Universal Windows Platform (UWP), the `ViewCellRenderer` class instantiates a native `DataTemplate`. For more information about the renderer and native control classes that `Xamarin.Forms` controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `ViewCell` and the corresponding native controls that implement it:



The rendering process can be taken advantage of to implement platform-specific customizations by creating a custom renderer for a `ViewCell` on each platform. The process for doing this is as follows:

1. [Create](#) a `Xamarin.Forms` custom cell.
2. [Consume](#) the custom cell from `Xamarin.Forms`.
3. [Create](#) the custom renderer for the cell on each platform.

Each item will now be discussed in turn, to implement a `NativeCell` renderer that takes advantage of a platform-specific layout for each cell hosted inside a `Xamarin.Forms ListView` control. This stops the `Xamarin.Forms` layout calculations from being repeatedly called during `ListView` scrolling.

## Creating the Custom Cell

A custom cell control can be created by subclassing the `ViewCell` class, as shown in the following code example:

```

public class NativeCell : ViewCell
{
    public static readonly BindableProperty NameProperty =
        BindableProperty.Create ("Name", typeof(string), typeof(NativeCell), "");

    public string Name {
        get { return (string)GetValue (NameProperty); }
        set { SetValue (NameProperty, value); }
    }

    public static readonly BindableProperty CategoryProperty =
        BindableProperty.Create ("Category", typeof(string), typeof(NativeCell), "");

    public string Category {
        get { return (string)GetValue (CategoryProperty); }
        set { SetValue (CategoryProperty, value); }
    }

    public static readonly BindableProperty ImageFilenameProperty =
        BindableProperty.Create ("ImageFilename", typeof(string), typeof(NativeCell), "");

    public string ImageFilename {
        get { return (string)GetValue (ImageFilenameProperty); }
        set { SetValue (ImageFilenameProperty, value); }
    }
}

```

The `NativeCell` class is created in the .NET Standard library project and defines the API for the custom cell. The custom cell exposes `Name`, `Category`, and `ImageFilename` properties that can be displayed through data binding. For more information about data binding, see [Data Binding Basics](#).

## Consuming the Custom Cell

The `NativeCell` custom cell can be referenced in Xaml in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the custom cell element. The following code example shows how the `NativeCell` custom cell can be consumed by a XAML page:

```

<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <ContentPage.Content>
        <StackLayout>
            <Label Text="Xamarin.Forms native cell" HorizontalTextAlignment="Center" />
            <ListView x:Name="listView" CachingStrategy="RecycleElement" ItemSelected="OnItemSelected">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <local:NativeCell Name="{Binding Name}" Category="{Binding Category}"
                            ImageFilename="{Binding ImageFilename}" />
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom cell.

The following code example shows how the `NativeCell` custom cell can be consumed by a C# page:

```

public class NativeCellPageCS : ContentPage
{
    ListView listView;

    public NativeCellPageCS()
    {
        listView = new ListView(ListViewCachingStrategy.RecycleElement)
        {
            ItemsSource = DataSource.GetList(),
            ItemTemplate = new DataTemplate(() =>
            {
                var nativeCell = new NativeCell();
                nativeCell.SetBinding(NativeCell.NameProperty, "Name");
                nativeCell.SetBinding(NativeCell.CategoryProperty, "Category");
                nativeCell.SetBinding(NativeCell.ImageFilenameProperty, "ImageFilename");

                return nativeCell;
            })
        };
    }

    switch (Device.RuntimePlatform)
    {
        case Device.iOS:
            Padding = new Thickness(0, 20, 0, 0);
            break;
        case Device.Android:
        case Device.UWP:
            Padding = new Thickness(0);
            break;
    }

    Content = new StackLayout
    {
        Children = {
            new Label { Text = "Xamarin.Forms native cell", HorizontalTextAlignment = TextAlignment.Center },
            listView
        }
    };
    listView.ItemSelected += OnItemSelected;
}
...
}

```

A Xamarin.Forms `ListView` control is used to display a list of data, which is populated through the `ItemsSource` property. The `RecycleElement` caching strategy attempts to minimize the `ListView` memory footprint and execution speed by recycling list cells. For more information, see [Caching Strategy](#).

Each row in the list contains three items of data – a name, a category, and an image filename. The layout of each row in the list is defined by the `DataTemplate` that's referenced through the `ListView.ItemTemplate` bindable property. The `DataTemplate` defines that each row of data in the list will be a `NativeCell` that displays its `Name`, `Category`, and `ImageFilename` properties through data binding. For more information about the `ListView` control, see [ListView](#).

A custom renderer can now be added to each application project to customize the platform-specific layout for each cell.

## Creating the Custom Renderer on each Platform

The process for creating the custom renderer class is as follows:

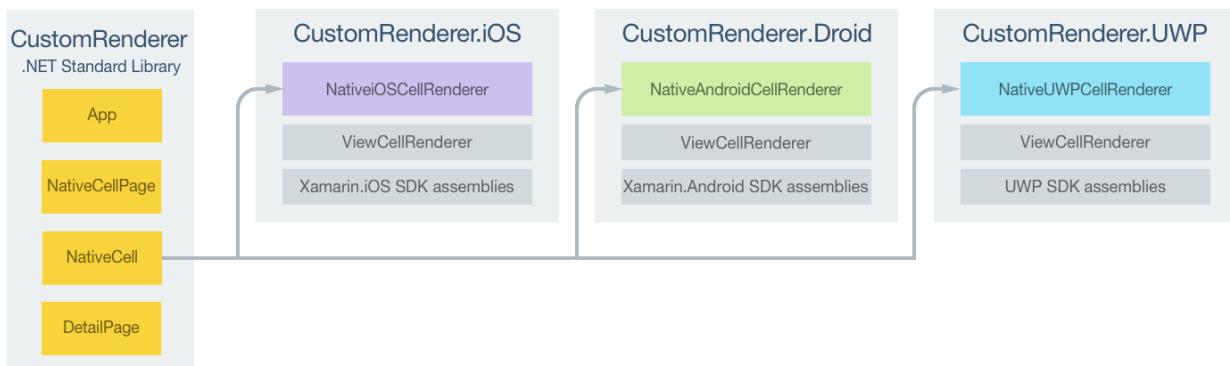
1. Create a subclass of the `ViewCellRenderer` class that renders the custom cell.

- Override the platform-specific method that renders the custom cell and write logic to customize it.
- Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the `Xamarin.Forms` custom cell. This attribute is used to register the custom renderer with `Xamarin.Forms`.

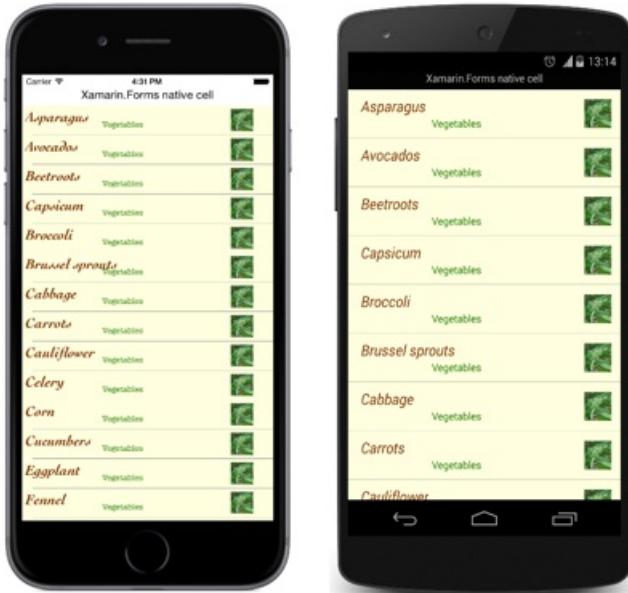
**NOTE**

For most `Xamarin.Forms` elements, it is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used. However, custom renderers are required in each platform project when rendering a `ViewCell` element.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `NativeCell` custom cell is rendered by platform-specific renderer classes, which all derive from the `ViewCellRenderer` class for each platform. This results in each `NativeCell` custom cell being rendered with platform-specific layout, as shown in the following screenshots:



The `ViewCellRenderer` class exposes platform-specific methods for rendering the custom cell. This is the `GetCell` method on the iOS platform, the `GetCellCore` method on the Android platform, and the `GetTemplate` method on UWP.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with `Xamarin.Forms`. The attribute takes two parameters – the type name of the `Xamarin.Forms` cell being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific custom renderer class.

## Creating the Custom Renderer on iOS

The following code example shows the custom renderer for the iOS platform:

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeiOSCellRenderer))]
namespace CustomRenderer.iOS
{
    public class NativeiOSCellRenderer : ViewCellRenderer
    {
        NativeiOSCell cell;

        public override UITableViewCell GetCell(Cell item, UITableView reusableCell, UITableView tv)
        {
            var nativeCell = (NativeCell)item;

            cell = reusableCell as NativeiOSCell;
            if (cell == null)
                cell = new NativeiOSCell(item.GetType().FullName, nativeCell);
            else
                cell.NativeCell.PropertyChanged -= OnNativeCellPropertyChanged;

            nativeCell.PropertyChanged += OnNativeCellPropertyChanged;
            cell.UpdateCell(nativeCell);
            return cell;
        }
        ...
    }
}
```

The `GetCell` method is called to build each cell to be displayed. Each cell is a `NativeiOSCell` instance, which defines the layout of the cell and its data. The operation of the `GetCell` method is dependent upon the `ListView` caching strategy:

- When the `ListView` caching strategy is `RetainElement`, the `GetCell` method will be invoked for each cell. A `NativeiOSCell` instance will be created for each `NativeCell` instance that's initially displayed on the screen. As the user scrolls through the `ListView`, `NativeiOSCell` instances will be re-used. For more information about iOS cell re-use, see [Cell Reuse](#).

### NOTE

This custom renderer code will perform some cell re-use even when the `ListView` is set to retain cells.

The data displayed by each `NativeiOSCell` instance, whether newly created or re-used, will be updated with the data from each `NativeCell` instance by the `UpdateCell` method.

### NOTE

The `OnNativeCellPropertyChanged` method will never be invoked when the `ListView` caching strategy is set to retain cells.

- When the `ListView` caching strategy is `RecycleElement`, the `GetCell` method will be invoked for each cell that's initially displayed on the screen. A `NativeiOSCell` instance will be created for each `NativeCell` instance that's initially displayed on the screen. The data displayed by each `NativeiOSCell` instance will be updated with the data from the `NativeCell` instance by the `UpdateCell` method. However, the `GetCell` method won't be invoked as the user scrolls through the `ListView`. Instead, the `NativeiOSCell` instances will be re-used. `PropertyChanged` events will be raised on the `NativeCell` instance when its data changes, and the `OnNativeCellPropertyChanged` event handler will update the data in each re-used `NativeiOSCell`.

instance.

The following code example shows the `OnNativeCellPropertyChanged` method that's invoked when a `PropertyChanged` event is raised:

```
namespace CustomRenderer.iOS
{
    public class NativeiOSCellRenderer : ViewCellRenderer
    {
        ...
        void OnNativeCellPropertyChanged(object sender, PropertyChangedEventArgs e)
        {
            var nativeCell = (NativeCell)sender;
            if (e.PropertyName == NativeCell.NameProperty.PropertyName)
            {
                cell.HeadingLabel.Text = nativeCell.Name;
            }
            else if (e.PropertyName == NativeCell.CategoryProperty.PropertyName)
            {
                cell.SubheadingLabel.Text = nativeCell.Category;
            }
            else if (e.PropertyName == NativeCell.ImageFilenameProperty.PropertyName)
            {
                cell.CellImageView.Image = cell.GetImage(nativeCell.ImageFilename);
            }
        }
    }
}
```

This method updates the data being displayed by re-used `NativeiOSCell` instances. A check for the property that's changed is made, as the method can be called multiple times.

The `NativeiOSCell` class defines the layout for each cell, and is shown in the following code example:

```

internal class NativeiOSCell : UITableViewCell, INativeElementView
{
    public UILabel HeadingLabel { get; set; }
    public UILabel SubheadingLabel { get; set; }
    public UIImageView CellImageView { get; set; }

    public NativeCell NativeCell { get; private set; }
    public Element Element => NativeCell;

    public NativeiOSCell(string cellId, NativeCell cell) : base(UITableViewCellStyle.Default, cellId)
    {
        NativeCell = cell;

        SelectionStyle = UITableViewCellSelectionStyle.Gray;
        ContentView.BackgroundColor = UIColor.FromRGB(255, 255, 224);
        CellImageView = new UIImageView();

        HeadingLabel = new UILabel()
        {
            Font = UIFont.FromName("Cochin-BoldItalic", 22f),
            TextColor = UIColor.FromRGB(127, 51, 0),
            BackgroundColor = UIColor.Clear
        };

        SubheadingLabel = new UILabel()
        {
            Font = UIFont.FromName("AmericanTypewriter", 12f),
            TextColor = UIColor.FromRGB(38, 127, 0),
            TextAlignment = UITextAlignment.Center,
            BackgroundColor = UIColor.Clear
        };

        ContentView.Add(HeadingLabel);
        ContentView.Add(SubheadingLabel);
        ContentView.Add(CellImageView);
    }

    public void UpdateCell(NativeCell cell)
    {
        HeadingLabel.Text = cell.Name;
        SubheadingLabel.Text = cell.Category;
        CellImageView.Image = GetImage(cell.ImageFilename);
    }

    public UIImage GetImage(string filename)
    {
        return (!string.IsNullOrWhiteSpace(filename)) ? UIImage.FromFile("Images/" + filename + ".jpg") : null;
    }

    public override void LayoutSubviews()
    {
        base.LayoutSubviews();

        HeadingLabel.Frame = new CGRect(5, 4, ContentView.Bounds.Width - 63, 25);
        SubheadingLabel.Frame = new CGRect(100, 18, 100, 20);
        CellImageView.Frame = new CGRect(ContentView.Bounds.Width - 63, 5, 33, 33);
    }
}

```

This class defines the controls used to render the cell's contents, and their layout. The class implements the `INativeElementView` interface, which is required when the `ListView` uses the `RecycleElement` caching strategy. This interface specifies that the class must implement the `Element` property, which should return the custom cell data for recycled cells.

The `NativeiOSCell` constructor initializes the appearance of the `HeadingLabel`, `SubheadingLabel`, and

`CellImageView` properties. These properties are used to display the data stored in the `NativeCell` instance, with the `UpdateCell` method being called to set the value of each property. In addition, when the `ListView` uses the `RecycleElement` caching strategy, the data displayed by the `HeadingLabel`, `SubheadingLabel`, and `CellImageView` properties can be updated by the `OnNativeCellPropertyChanged` method in the custom renderer.

Cell layout is performed by the `LayoutSubviews` override, which sets the coordinates of `HeadingLabel`, `SubheadingLabel`, and `CellImageView` within the cell.

## Creating the Custom Renderer on Android

The following code example shows the custom renderer for the Android platform:

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeAndroidCellRenderer))]
namespace CustomRenderer.Droid
{
    public class NativeAndroidCellRenderer : ViewCellRenderer
    {
        NativeAndroidCell cell;

        protected override Android.Views.View GetCellCore(Cell item, Android.Views.View convertView, ViewGroup parent, Context context)
        {
            var nativeCell = (NativeCell)item;
            Console.WriteLine("\t\t" + nativeCell.Name);

            cell = convertView as NativeAndroidCell;
            if (cell == null)
            {
                cell = new NativeAndroidCell(context, nativeCell);
            }
            else
            {
                cell.NativeCell.PropertyChanged -= OnNativeCellPropertyChanged;
            }

            nativeCell.PropertyChanged += OnNativeCellPropertyChanged;

            cell.UpdateCell(nativeCell);
            return cell;
        }
        ...
    }
}
```

The `GetCellCore` method is called to build each cell to be displayed. Each cell is a `NativeAndroidCell` instance, which defines the layout of the cell and its data. The operation of the `GetCellCore` method is dependent upon the `ListView` caching strategy:

- When the `ListView` caching strategy is `RetainElement`, the `GetCellCore` method will be invoked for each cell. A `NativeAndroidCell` will be created for each `NativeCell` instance that's initially displayed on the screen. As the user scrolls through the `ListView`, `NativeAndroidCell` instances will be re-used. For more information about Android cell re-use, see [Row View Re-use](#).

### NOTE

Note that this custom renderer code will perform some cell re-use even when the `ListView` is set to retain cells.

The data displayed by each `NativeAndroidCell` instance, whether newly created or re-used, will be updated with the data from each `NativeCell` instance by the `UpdateCell` method.

#### NOTE

Note that while the `OnNativeCellPropertyChanged` method will be invoked when the `ListView` is set to retain cells, it will not update the `NativeAndroidCell` property values.

- When the `ListView` caching strategy is `RecycleElement`, the `GetCellCore` method will be invoked for each cell that's initially displayed on the screen. A `NativeAndroidCell` instance will be created for each `NativeCell` instance that's initially displayed on the screen. The data displayed by each `NativeAndroidCell` instance will be updated with the data from the `NativeCell` instance by the `UpdateCell` method. However, the `GetCellCore` method won't be invoked as the user scrolls through the `ListView`. Instead, the `NativeAndroidCell` instances will be re-used. `PropertyChanged` events will be raised on the `NativeCell` instance when its data changes, and the `OnNativeCellPropertyChanged` event handler will update the data in each re-used `NativeAndroidCell` instance.

The following code example shows the `OnNativeCellPropertyChanged` method that's invoked when a `PropertyChanged` event is raised:

```
namespace CustomRenderer.Droid
{
    public class NativeAndroidCellRenderer : ViewCellRenderer
    {
        ...
        void OnNativeCellPropertyChanged(object sender, PropertyChangedEventArgs e)
        {
            var nativeCell = (NativeCell)sender;
            if (e.PropertyName == NativeCell.NameProperty.PropertyName)
            {
                cell.HeadingTextView.Text = nativeCell.Name;
            }
            else if (e.PropertyName == NativeCell.CategoryProperty.PropertyName)
            {
                cell.SubheadingTextView.Text = nativeCell.Category;
            }
            else if (e.PropertyName == NativeCell.ImageFilenameProperty.PropertyName)
            {
                cell.SetImage(nativeCell.ImageFilename);
            }
        }
    }
}
```

This method updates the data being displayed by re-used `NativeAndroidCell` instances. A check for the property that's changed is made, as the method can be called multiple times.

The `NativeAndroidCell` class defines the layout for each cell, and is shown in the following code example:

```

internal class NativeAndroidCell : LinearLayout, INativeElementView
{
    public TextView HeadingTextView { get; set; }
    public TextView SubheadingTextView { get; set; }
    public ImageView ImageView { get; set; }

    public NativeCell NativeCell { get; private set; }
    public Element Element => NativeCell;

    public NativeAndroidCell(Context context, NativeCell cell) : base(context)
    {
        NativeCell = cell;

        var view = (context as Activity).LayoutInflater.Inflate(Resource.Layout.NativeAndroidCell, null);
        HeadingTextView = view.FindViewById<TextView>(Resource.Id.HeadingText);
        SubheadingTextView = view.FindViewById<TextView>(Resource.Id.SubheadingText);
        ImageView = view.FindViewById<ImageView>(Resource.Id.Image);

        AddView(view);
    }

    public void UpdateCell(NativeCell cell)
    {
        HeadingTextView.Text = cell.Name;
        SubheadingTextView.Text = cell.Category;

        // Dispose of the old image
        if (ImageView.Drawable != null)
        {
            using (var image = ImageView.Drawable as BitmapDrawable)
            {
                if (image != null)
                {
                    if (image.Bitmap != null)
                    {
                        image.Bitmap.Dispose();
                    }
                }
            }
        }

        SetImage(cell.ImageFilename);
    }

    public void SetImage(string filename)
    {
        if (!string.IsNullOrWhiteSpace(filename))
        {
            // Display new image
            Context.Resources.GetBitmapAsync(filename).ContinueWith((t) =>
            {
                var bitmap = t.Result;
                if (bitmap != null)
                {
                    ImageView.SetImageBitmap(bitmap);
                    bitmap.Dispose();
                }
            }, TaskScheduler.FromCurrentSynchronizationContext());
        }
        else
        {
            // Clear the image
            ImageView.SetImageBitmap(null);
        }
    }
}

```

This class defines the controls used to render the cell's contents, and their layout. The class implements the `INativeElementView` interface, which is required when the `ListView` uses the `RecycleElement` caching strategy. This interface specifies that the class must implement the `Element` property, which should return the custom cell data for recycled cells.

The `NativeAndroidCell` constructor inflates the `NativeAndroidCell` layout, and initializes the `HeadingTextView`, `SubheadingTextView`, and `ImageView` properties to the controls in the inflated layout. These properties are used to display the data stored in the `NativeCell` instance, with the `UpdateCell` method being called to set the value of each property. In addition, when the `ListView` uses the `RecycleElement` caching strategy, the data displayed by the `HeadingTextView`, `SubheadingTextView`, and `ImageView` properties can be updated by the `OnNativeCellPropertyChanged` method in the custom renderer.

The following code example shows the layout definition for the `NativeAndroidCell.axml` layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:background="@drawable/CustomSelector">
    <LinearLayout
        android:id="@+id/Text"
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="10dip">
        <TextView
            android:id="@+id/HeadingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dip"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/SubheadingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="14dip"
            android:textColor="#FF267F00"
            android:paddingLeft="100dip" />
    </LinearLayout>
    <ImageView
        android:id="@+id/Image"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:padding="5dp"
        android:src="@drawable/icon"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

This layout specifies that two `TextView` controls and an `ImageView` control are used to display the cell's content. The two `TextView` controls are vertically oriented within a `LinearLayout` control, with all the controls being contained within a `RelativeLayout`.

## Creating the Custom Renderer on UWP

The following code example shows the custom renderer for UWP:

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeUWPCellRenderer))]
namespace CustomRenderer.UWP
{
    public class NativeUWPCellRenderer : ViewCellRenderer
    {
        public override Windows.UI.Xaml.DataTemplate GetTemplate(Cell cell)
        {
            return App.Current.Resources["ListViewItemTemplate"] as Windows.UI.Xaml.DataTemplate;
        }
    }
}
```

The `GetTemplate` method is called to return the cell to be rendered for each row of data in the list. It creates a `DataTemplate` for each `NativeCell` instance that will be displayed on the screen, with the `DataTemplate` defining the appearance and contents of the cell.

The `DataTemplate` is stored in the application-level resource dictionary, and is shown in the following code example:

```
<DataTemplate x:Key="ListViewItemTemplate">
    <Grid Background="LightYellow">
        <Grid.Resources>
            <local:ConcatImageExtensionConverter x:Name="ConcatImageExtensionConverter" />
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.20*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.ColumnSpan="2" Foreground="#7F3300" FontStyle="Italic" FontSize="22"
VerticalAlignment="Top" Text="{Binding Name}" />
        <TextBlock Grid.RowSpan="2" Grid.Column="1" Foreground="#267F00" FontWeight="Bold" FontSize="12"
VerticalAlignment="Bottom" Text="{Binding Category}" />
        <Image Grid.RowSpan="2" Grid.Column="2" HorizontalAlignment="Left" VerticalAlignment="Center" Source="
{Binding ImageFilename, Converter={StaticResource ConcatImageExtensionConverter}}" Width="50" Height="50" />
        <Line Grid.Row="1" Grid.ColumnSpan="3" X1="0" X2="1" Margin="30,20,0,0" StrokeThickness="1"
Stroke="LightGray" Stretch="Fill" VerticalAlignment="Bottom" />
    </Grid>
</DataTemplate>
```

The `DataTemplate` specifies the controls used to display the contents of the cell, and their layout and appearance. Two `TextBlock` controls and an `Image` control are used to display the cell's content through data binding. In addition, an instance of the `ConcatImageExtensionConverter` is used to concatenate the `.jpg` file extension to each image file name. This ensures that the `Image` control can load and render the image when its `Source` property is set.

## Summary

This article has demonstrated how to create a custom renderer for a `ViewCell` that's hosted inside a `Xamarin.Forms ListView` control. This stops the `Xamarin.Forms` layout calculations from being repeatedly called during `ListView` scrolling.

## Related Links

- [ListView Performance](#)

- [CustomRendererViewCell \(sample\)](#)

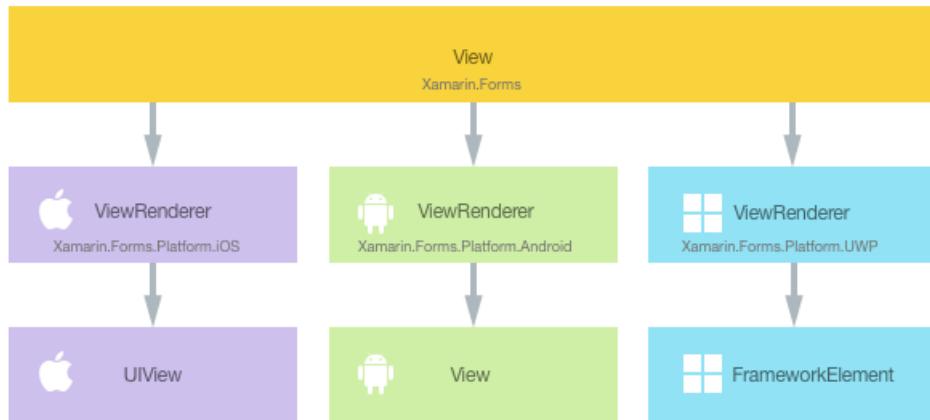
# Implementing a View

7/12/2018 • 9 minutes to read • [Edit Online](#)

Xamarin.Forms custom user interface controls should derive from the `View` class, which is used to place layouts and controls on the screen. This article demonstrates how to create a custom renderer for a Xamarin.Forms custom control that's used to display a preview video stream from the device's camera.

Every Xamarin.Forms view has an accompanying renderer for each platform that creates an instance of a native control. When a `View` is rendered by a Xamarin.Forms application in iOS, the `ViewRenderer` class is instantiated, which in turn instantiates a native `UIView` control. On the Android platform, the `ViewRenderer` class instantiates a native `View` control. On the Universal Windows Platform (UWP), the `ViewRenderer` class instantiates a native `FrameworkElement` control. For more information about the renderer and native control classes that Xamarin.Forms controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `View` and the corresponding native controls that implement it:



The rendering process can be used to implement platform-specific customizations by creating a custom renderer for a `View` on each platform. The process for doing this is as follows:

1. [Create](#) a Xamarin.Forms custom control.
2. [Consume](#) the custom control from Xamarin.Forms.
3. [Create](#) the custom renderer for the control on each platform.

Each item will now be discussed in turn, to implement a `CameraPreview` renderer that displays a preview video stream from the device's camera. Tapping on the video stream will stop and start it.

## Creating the Custom Control

A custom control can be created by subclassing the `View` class, as shown in the following code example:

```

public class CameraPreview : View
{
    public static readonly BindableProperty CameraProperty = BindableProperty.Create (
        propertyName: "Camera",
        returnType: typeof(CameraOptions),
        declaringType: typeof(CameraPreview),
        defaultValue: CameraOptions.Rear);

    public CameraOptions Camera {
        get { return (CameraOptions)GetValue (CameraProperty); }
        set { SetValue (CameraProperty, value); }
    }
}

```

The `CameraPreview` custom control is created in the portable class library (PCL) project and defines the API for the control. The custom control exposes a `Camera` property that's used for controlling whether the video stream should be displayed from the front or rear camera on the device. If a value isn't specified for the `Camera` property when the control is created, it defaults to specifying the rear camera.

## Consuming the Custom Control

The `CameraPreview` custom control can be referenced in XAML in the PCL project by declaring a namespace for its location and using the namespace prefix on the custom control element. The following code example shows how the `CameraPreview` custom control can be consumed by a XAML page:

```

<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    <ContentPage.Content>
        <StackLayout>
            <Label Text="Camera Preview:" />
            <local:CameraPreview Camera="Rear"
                HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom control.

The following code example shows how the `CameraPreview` custom control can be consumed by a C# page:

```

public class MainPageCS : ContentPage
{
    public MainPageCS ()
    {
        ...
        Content = new StackLayout {
            Children = {
                new Label { Text = "Camera Preview:" },
                new CameraPreview {
                    Camera = CameraOptions.Rear,
                    HorizontalOptions = LayoutOptions.FillAndExpand,
                    VerticalOptions = LayoutOptions.FillAndExpand
                }
            }
        };
    }
}

```

An instance of the `CameraPreview` custom control will be used to display the preview video stream from the device's camera. Aside from optionally specifying a value for the `Camera` property, customization of the control will be carried out in the custom renderer.

A custom renderer can now be added to each application project to create platform-specific camera preview controls.

## Creating the Custom Renderer on each Platform

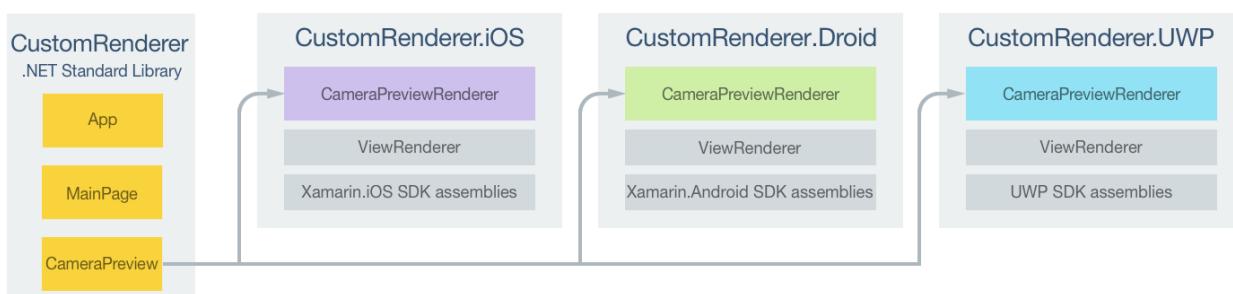
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `ViewRenderer<T1, T2>` class that renders the custom control. The first type argument should be the custom control the renderer is for, in this case `CameraPreview`. The second type argument should be the native control that will implement the custom control.
2. Override the `OnElementChanged` method that renders the custom control and write logic to customize it. This method is called when the corresponding Xamarin.Forms control is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms custom control. This attribute is used to register the custom renderer with Xamarin.Forms.

### NOTE

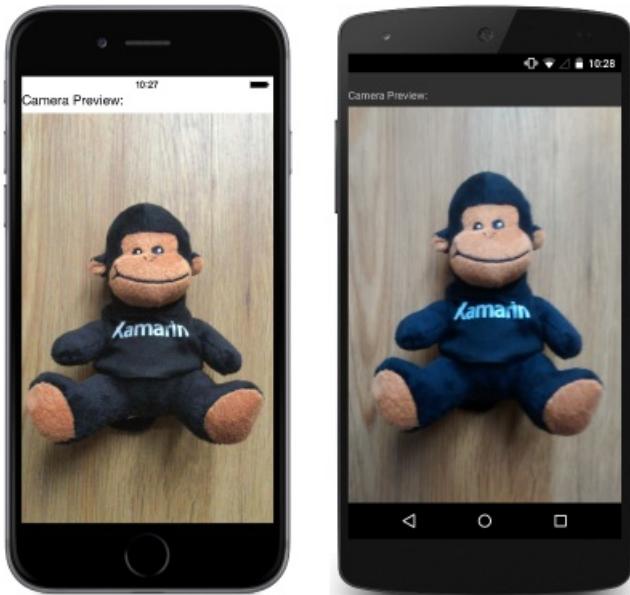
For most Xamarin.Forms elements, it is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used. However, custom renderers are required in each platform project when rendering a `View` element.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `CameraPreview` custom control is rendered by platform-specific renderer classes, which all derive from the

`ViewRenderer` class for each platform. This results in each `CameraPreview` custom control being rendered with platform-specific controls, as shown in the following screenshots:



The `ViewRenderer` class exposes the `OnElementChanged` method, which is called when the Xamarin.Forms custom control is created to render the corresponding native control. This method takes an `ElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the Xamarin.Forms element that the renderer *was* attached to, and the Xamarin.Forms element that the renderer *is* attached to, respectively. In the sample application, the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `CameraPreview` instance.

An overridden version of the `OnElementChanged` method, in each platform-specific renderer class, is the place to perform the native control instantiation and customization. The `SetNativeControl` method should be used to instantiate the native control, and this method will also assign the control reference to the `Control` property. In addition, a reference to the Xamarin.Forms control that's being rendered can be obtained through the `Element` property.

In some circumstances, the `OnElementChanged` method can be called multiple times. Therefore, to prevent memory leaks, care must be taken when instantiating a new native control. The approach to use when instantiating a new native control in a custom renderer is shown in the following code example:

```
protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
    base.OnElementChanged (e);

    if (Control == null) {
        // Instantiate the native control and assign it to the Control property with
        // the SetNativeControl method
    }

    if (e.OldElement != null) {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null) {
        // Configure the control and subscribe to event handlers
    }
}
```

A new native control should only be instantiated once, when the `Control` property is `null`. The control should only be configured and event handlers subscribed to when the custom renderer is attached to a new

Xamarin.Forms element. Similarly, any event handlers that were subscribed to should only be unsubscribed from when the element that the renderer is attached to changes. Adopting this approach will help to create a performant custom renderer that doesn't suffer from memory leaks.

Each custom renderer class is decorated with an `[ExportRenderer]` attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms custom control being rendered, and the type name of the custom renderer. The `[assembly]` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific custom renderer class.

## Creating the Custom Renderer on iOS

The following code example shows the custom renderer for the iOS platform:

```
[assembly: ExportRenderer (typeof(CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.iOS
{
    public class CameraPreviewRenderer : ViewRenderer<CameraPreview, UICameraPreview>
    {
        UICameraPreview uiCameraPreview;

        protected override void OnElementChanged (ElementChangedEventArgs<CameraPreview> e)
        {
            base.OnElementChanged (e);

            if (Control == null) {
                uiCameraPreview = new UICameraPreview (e.NewElement.Camera);
                SetNativeControl (uiCameraPreview);
            }
            if (e.OldElement != null) {
                // Unsubscribe
                uiCameraPreview.Tapped -= OnCameraPreviewTapped;
            }
            if (e.NewElement != null) {
                // Subscribe
                uiCameraPreview.Tapped += OnCameraPreviewTapped;
            }
        }

        void OnCameraPreviewTapped (object sender, EventArgs e)
        {
            if (uiCameraPreview.IsPreviewing) {
                uiCameraPreview.CaptureSession.StopRunning ();
                uiCameraPreview.IsPreviewing = false;
            } else {
                uiCameraPreview.CaptureSession.StartRunning ();
                uiCameraPreview.IsPreviewing = true;
            }
        }
        ...
    }
}
```

Provided that the `Control` property is `null`, the `SetNativeControl` method is called to instantiate a new `UICameraPreview` control and to assign a reference to it to the `Control` property. The `uiCameraPreview` control is a platform-specific custom control that uses the `AVCapture` APIs to provide the preview stream from the camera. It exposes a `Tapped` event that's handled by the `OnCameraPreviewTapped` method to stop and start the video preview when it's tapped. The `Tapped` event is subscribed to when the custom renderer is attached to a new Xamarin.Forms element, and unsubscribed from only when the element the renderer is attached to changes.

## Creating the Custom Renderer on Android

The following code example shows the custom renderer for the Android platform:

```

[assembly: ExportRenderer(typeof(CustomRenderer.CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.Droid
{
    public class CameraPreviewRenderer : ViewRenderer<CustomRenderer.CameraPreview,
    CustomRenderer.Droid.CameraPreview>
    {
        CameraPreview cameraPreview;

        public CameraPreviewRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<CustomRenderer.CameraPreview> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                cameraPreview = new CameraPreview(Context);
                SetNativeControl(cameraPreview);
            }

            if (e.OldElement != null)
            {
                // Unsubscribe
                cameraPreview.Click -= OnCameraPreviewClicked;
            }
            if (e.NewElement != null)
            {
                Control.Preview = Camera.Open((int)e.NewElement.Camera);

                // Subscribe
                cameraPreview.Click += OnCameraPreviewClicked;
            }
        }

        void OnCameraPreviewClicked(object sender, EventArgs e)
        {
            if (cameraPreview.IsPreviewing)
            {
                cameraPreview.Preview.StopPreview();
                cameraPreview.IsPreviewing = false;
            }
            else
            {
                cameraPreview.Preview.StartPreview();
                cameraPreview.IsPreviewing = true;
            }
        }
        ...
    }
}

```

Provided that the `Control` property is `null`, the `SetNativeControl` method is called to instantiate a new `CameraPreview` control and assign a reference to it to the `Control` property. The `CameraPreview` control is a platform-specific custom control that uses the `Camera` API to provide the preview stream from the camera. The `CameraPreview` control is then configured, provided that the custom renderer is attached to a new `Xamarin.Forms` element. This configuration involves creating a new native `Camera` object to access a particular hardware camera, and registering an event handler to process the `Click` event. In turn this handler will stop and start the video preview when it's tapped. The `Click` event is unsubscribed from if the `Xamarin.Forms` element the renderer is attached to changes.

## Creating the Custom Renderer on UWP

The following code example shows the custom renderer for UWP:

```
[assembly: ExportRenderer(typeof(CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.UWP
{
    public class CameraPreviewRenderer : ViewRenderer<CameraPreview, Windows.UI.Xaml.Controls.CaptureElement>
    {
        ...
        CaptureElement _captureElement;
        bool _isPreviewing;

        protected override void OnElementChanged(ElementChangedEventArgs<CameraPreview> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                ...
                _captureElement = new CaptureElement();
                _captureElement.Stretch = Stretch.UniformToFill;

                SetupCamera();
                SetNativeControl(_captureElement);
            }
            if (e.OldElement != null)
            {
                // Unsubscribe
                Tapped -= OnCameraPreviewTapped;
                ...
            }
            if (e.NewElement != null)
            {
                // Subscribe
                Tapped += OnCameraPreviewTapped;
            }
        }

        async void OnCameraPreviewTapped(object sender, TappedRoutedEventArgs e)
        {
            if (_isPreviewing)
            {
                await StopPreviewAsync();
            }
            else
            {
                await StartPreviewAsync();
            }
        }
        ...
    }
}
```

Provided that the `Control` property is `null`, a new `CaptureElement` is instantiated and the `SetupCamera` method is called, which uses the `MediaCapture` API to provide the preview stream from the camera. The `SetNativeControl` method is then called to assign a reference to the `CaptureElement` instance to the `Control` property. The `CaptureElement` control exposes a `Tapped` event that's handled by the `OnCameraPreviewTapped` method to stop and start the video preview when it's tapped. The `Tapped` event is subscribed to when the custom renderer is attached to a new `Xamarin.Forms` element, and unsubscribed from only when the element the renderer is attached to changes.

**NOTE**

It's important to stop and dispose of the objects that provide access to the camera in a UWP application. Failure to do so can interfere with other applications that attempt to access the device's camera. For more information, see [Display the camera preview](#).

## Summary

This article has demonstrated how to create a custom renderer for a Xamarin.Forms custom control that's used to display a preview video stream from the device's camera. Xamarin.Forms custom user interface controls should derive from the `View` class, which is used to place layouts and controls on the screen.

## Related Links

- [CustomRendererView \(sample\)](#)

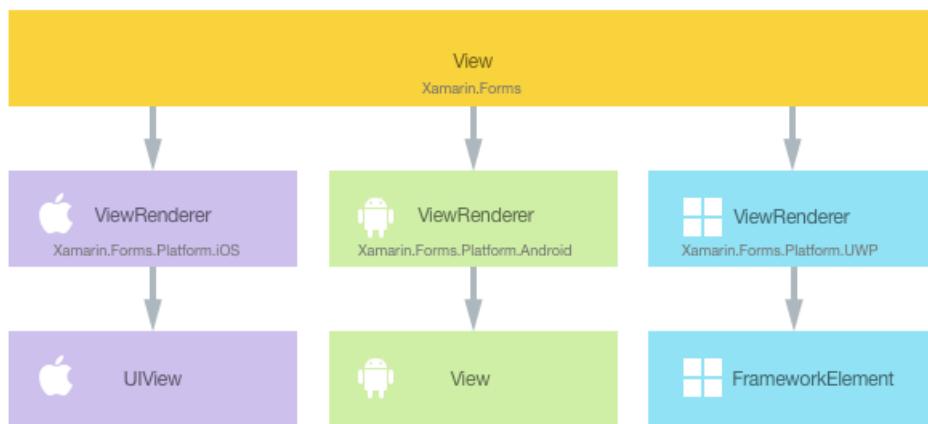
# Implementing a HybridWebView

10/19/2018 • 16 minutes to read • [Edit Online](#)

Xamarin.Forms custom user interface controls should derive from the `View` class, which is used to place layouts and controls on the screen. This article demonstrates how to create a custom renderer for a `HybridWebView` custom control, which demonstrates how to enhance the platform-specific web controls to allow C# code to be invoked from JavaScript.

Every Xamarin.Forms view has an accompanying renderer for each platform that creates an instance of a native control. When a `View` is rendered by a Xamarin.Forms application in iOS, the `ViewRenderer` class is instantiated, which in turn instantiates a native `UIWebView` control. On the Android platform, the `ViewRenderer` class instantiates a `View` control. On the Universal Windows Platform (UWP), the `ViewRenderer` class instantiates a native `FrameworkElement` control. For more information about the renderer and native control classes that Xamarin.Forms controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `View` and the corresponding native controls that implement it:



The rendering process can be used to implement platform-specific customizations by creating a custom renderer for a `View` on each platform. The process for doing this is as follows:

1. [Create](#) the `HybridWebView` custom control.
2. [Consume](#) the `HybridWebView` from Xamarin.Forms.
3. [Create](#) the custom renderer for the `HybridWebView` on each platform.

Each item will now be discussed in turn to implement a `HybridWebView` renderer that enhances the platform-specific web controls to allow C# code to be invoked from JavaScript. The `HybridWebView` instance will be used to display an HTML page that asks the user to enter their name. Then, when the user clicks an HTML button, a JavaScript function will invoke a C# `Action` that displays a pop-up containing the users name.

For more information about the process for invoking C# from JavaScript, see [Invoking C# from JavaScript](#). For more information about the HTML page, see [Creating the Web Page](#).

## Creating the HybridWebView

The `HybridWebView` custom control can be created by subclassing the `View` class, as shown in the following code example:

```

public class HybridWebView : View
{
    Action<string> action;
    public static readonly BindableProperty UriProperty = BindableProperty.Create (
        propertyName: "Uri",
        returnType: typeof(string),
        declaringType: typeof(HybridWebView),
        defaultValue: default(string));

    public string Uri {
        get { return (string)GetValue (UriProperty); }
        set { SetValue (UriProperty, value); }
    }

    public void RegisterAction (Action<string> callback)
    {
        action = callback;
    }

    public void Cleanup ()
    {
        action = null;
    }

    public void InvokeAction (string data)
    {
        if (action == null || data == null) {
            return;
        }
        action.Invoke (data);
    }
}

```

The `HybridWebView` custom control is created in the .NET Standard library project and defines the following API for the control:

- A `Uri` property that specifies the address of the web page to be loaded.
- A `RegisterAction` method that registers an `Action` with the control. The registered action will be invoked from JavaScript contained in the HTML file referenced through the `Uri` property.
- A `CleanUp` method that removes the reference to the registered `Action`.
- An `InvokeAction` method that invokes the registered `Action`. This method will be called from a custom renderer in each platform-specific project.

## Consuming the HybridWebView

The `HybridWebView` custom control can be referenced in XAML in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the custom control. The following code example shows how the `HybridWebView` custom control can be consumed by a XAML page:

```

<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    x:Class="CustomRenderer.HybridWebViewPage"
    Padding="0,20,0,0">
    <ContentPage.Content>
        <local:HybridWebView x:Name="hybridWebView" Uri="index.html"
            HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" />
    </ContentPage.Content>
</ContentPage>

```

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must

match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom control.

The following code example shows how the `HybridWebView` custom control can be consumed by a C# page:

```
public class HybridWebViewPageCS : ContentPage
{
    public HybridWebViewPageCS ()
    {
        var hybridWebView = new HybridWebView {
            Uri = "index.html",
            HorizontalOptions = LayoutOptions.FillAndExpand,
            VerticalOptions = LayoutOptions.FillAndExpand
        };
        ...
        Padding = new Thickness (0, 20, 0, 0);
        Content = hybridWebView;
    }
}
```

The `HybridWebView` instance will be used to display a native web control on each platform. It's `Uri` property is set to an HTML file that is stored in each platform-specific project, and which will be displayed by the native web control. The rendered HTML asks the user to enter their name, with a JavaScript function invoking a C# `Action` in response to an HTML button click.

The `HybridWebViewPage` registers the action to be invoked from JavaScript, as shown in the following code example:

```
public partial class HybridWebViewPage : ContentPage
{
    public HybridWebViewPage ()
    {
        ...
        hybridWebView.RegisterAction (data => DisplayAlert ("Alert", "Hello " + data, "OK"));
    }
}
```

This action calls the `DisplayAlert` method to display a modal pop-up that presents the name entered in the HTML page displayed by the `HybridWebView` instance.

A custom renderer can now be added to each application project to enhance the platform-specific web controls by allowing C# code to be invoked from JavaScript.

## Creating the Custom Renderer on each Platform

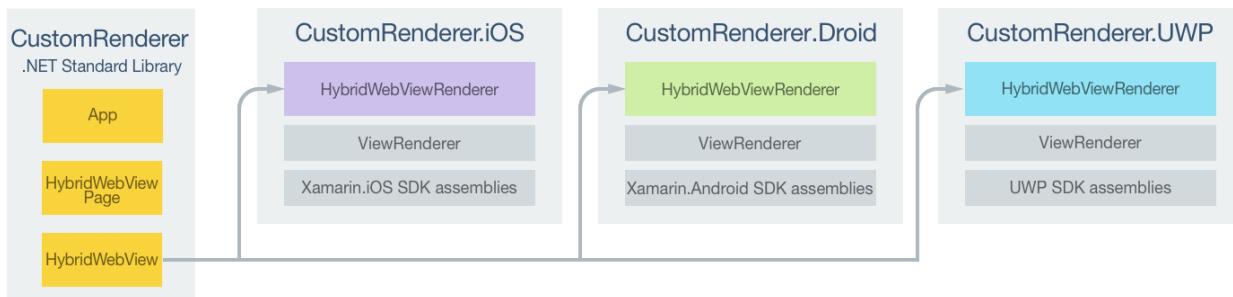
The process for creating the custom renderer class is as follows:

1. Create a subclass of the `ViewRenderer<T1, T2>` class that renders the custom control. The first type argument should be the custom control the renderer is for, in this case `HybridWebView`. The second type argument should be the native control that will implement the custom view.
2. Override the `OnElementChanged` method that renders the custom control and write logic to customize it. This method is called when the corresponding Xamarin.Forms custom control is created.
3. Add an `ExportRenderer` attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms custom control. This attribute is used to register the custom renderer with Xamarin.Forms.

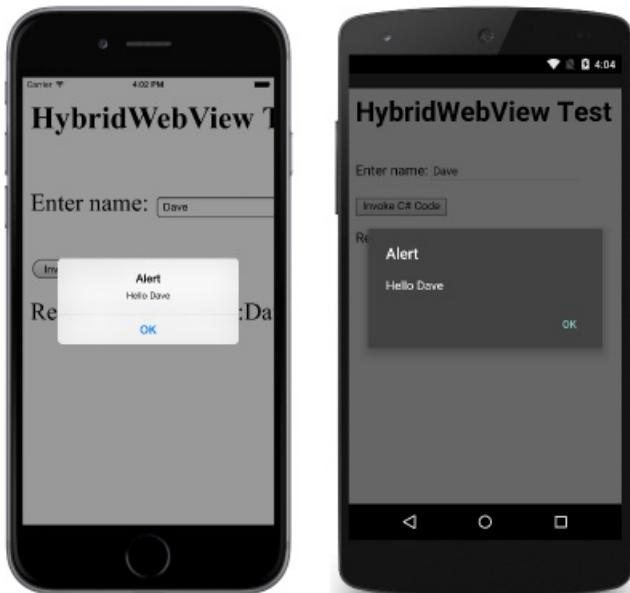
## NOTE

For most Xamarin.Forms elements, it is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used. However, custom renderers are required in each platform project when rendering a `View` element.

The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



The `HybridWebView` custom control is rendered by platform-specific renderer classes, which all derive from the `ViewRenderer` class for each platform. This results in each `HybridWebView` custom control being rendered with platform-specific web controls, as shown in the following screenshots:



The `ViewRenderer` class exposes the `OnElementChanged` method, which is called when the Xamarin.Forms custom control is created to render the corresponding native web control. This method takes an `ElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the Xamarin.Forms element that the renderer was attached to, and the Xamarin.Forms element that the renderer is attached to, respectively. In the sample application the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `HybridWebView` instance.

An overridden version of the `OnElementChanged` method, in each platform-specific renderer class, is the place to perform the native web control instantiation and customization. The `SetNativeControl` method should be used to instantiate the native web control, and this method will also assign the control reference to the `Control` property. In addition, a reference to the Xamarin.Forms control that's being rendered can be obtained through the `Element` property.

In some circumstances the `OnElementChanged` method can be called multiple times. Therefore, to prevent memory leaks, care must be taken when instantiating a new native control. The approach to use when instantiating a new

native control in a custom renderer is shown in the following code example:

```
protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
    base.OnElementChanged (e);

    if (Control == null) {
        // Instantiate the native control and assign it to the Control property with
        // the SetNativeControl method
    }

    if (e.OldElement != null) {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null) {
        // Configure the control and subscribe to event handlers
    }
}
```

A new native control should only be instantiated once, when the `Control` property is `null`. The control should only be configured and event handlers subscribed to when the custom renderer is attached to a new Xamarin.Forms element. Similarly, any event handlers that were subscribed to should only be unsubscribed from when the element the renderer is attached to changes. Adopting this approach will help to create a performant custom renderer that doesn't suffer from memory leaks.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms custom control being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the structure of the web page loaded by each native web control, the process for invoking C# from JavaScript, and the implementation of this in each platform-specific custom renderer class.

## Creating the Web Page

The following code example shows the web page that will be displayed by the `HybridWebView` custom control:

```

<html>
<body>
<script src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
<h1>HybridWebView Test</h1>
<br/>
Enter name: <input type="text" id="name">
<br/>
<br/>
<button type="button" onclick="javascript:invokeCSCode($('#name').val());">Invoke C# Code</button>
<br/>
<p id="result">Result:</p>
<script type="text/javascript">
function log(str)
{
    $('#result').text($('#result').text() + " " + str);
}

function invokeCSCode(data) {
    try {
        log("Sending Data:" + data);
        invokeCSharpAction(data);
    }
    catch (err){
        log(err);
    }
}
</script>
</body>
</html>

```

The web page allows a user to enter their name in an `input` element, and provides a `button` element that will invoke C# code when clicked. The process for achieving this is as follows:

- When the user clicks on the `button` element, the `invokeCSCode` JavaScript function is called, with the value of the `input` element being passed to the function.
- The `invokeCSCode` function calls the `log` function to display the data it is sending to the C# `Action`. It then calls the `invokeCSharpAction` method to invoke the C# `Action`, passing the parameter received from the `input` element.

The `invokeCSharpAction` JavaScript function is not defined in the web page, and will be injected into it by each custom renderer.

## Invoking C# from JavaScript

The process for invoking C# from JavaScript is identical on each platform:

- The custom renderer creates a native web control and loads the HTML file specified by the `HybridWebView.Uri` property.
- Once the web page is loaded, the custom renderer injects the `invokeCSharpAction` JavaScript function into the web page.
- When the user enters their name and clicks on the HTML `button` element, the `invokeCSCode` function is invoked, which in turn invokes the `invokeCSharpAction` function.
- The `invokeCSharpAction` function invokes a method in the custom renderer, which in turn invokes the `HybridWebView.InvokeAction` method.
- The `HybridWebView.InvokeAction` method invokes the registered `Action`.

The following sections will discuss how this process is implemented on each platform.

## Creating the Custom Renderer on iOS

The following code example shows the custom renderer for the iOS platform:

```

[assembly: ExportRenderer (typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.iOS
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, WKWebView>, IWKScriptMessageHandler
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.webkit.messageHandlers.invokeAction.postMessage(data);}";
        WKUserContentController userController;

        protected override void OnElementChanged (ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged (e);

            if (Control == null) {
                userController = new WKUserContentController ();
                var script = new WKUserScript (new NSString (JavaScriptFunction),
WKUserScriptInjectionTime.AtDocumentEnd, false);
                userController.AddUserScript (script);
                userController.AddScriptMessageHandler (this, "invokeAction");

                var config = new WKWebViewConfiguration { UserContentController = userController };
                var webView = new WKWebView (Frame, config);
                SetNativeControl (webView);
            }
            if (e.OldElement != null) {
                userController.RemoveAllUserScripts ();
                userController.RemoveScriptMessageHandler ("invokeAction");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup ();
            }
            if (e.NewElement != null) {
                string fileName = Path.Combine (NSBundle.MainBundle.BundlePath, string.Format ("Content/{0}",
Element.Uri));
                Control.LoadRequest (new NSUrlRequest (new NSUrl (fileName, false)));
            }
        }

        public void DidReceiveScriptMessage (WKUserContentController userContentController, WKScriptMessage
message)
        {
            Element.InvokeAction (message.Body.ToString ());
        }
    }
}

```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native `WKWebView` control, and the `invokeCSharpAction` JavaScript function is injected into the web page. Once the user enters their name and clicks the HTML `button` element, the `invokeCSharpAction` JavaScript function is executed, with the `DidReceiveScriptMessage` method being called after a message is received from the web page. In turn, this method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

This functionality is achieved as follows:

- Provided that the `Control` property is `null`, the following operations are carried out:
  - A `WKUserContentController` instance is created, which allows posting messages and injecting user scripts into a web page.
  - A `WKUserScript` instance is created to inject the `invokeCSharpAction` JavaScript function into the web page after the web page is loaded.
  - The `WKUserContentController.AddScript` method adds the `WKUserScript` instance to the content controller.

- The `WKUserContentController.AddScriptMessageHandler` method adds a script message handler named `invokeAction` to the `WKUserContentController` instance, which will cause the JavaScript function `window.webkit.messageHandlers.invokeAction.postMessage(data)` to be defined in all frames in all web views that will use the `WKUserContentController` instance.
  - A `WKWebViewConfiguration` instance is created, with the `WKUserContentController` instance being set as the content controller.
  - A `WKWebView` control is instantiated, and the `SetNativeControl` method is called to assign a reference to the `WKWebView` control to the `control` property.
- Provided that the custom renderer is attached to a new Xamarin.Forms element:
    - The `WKWebView.LoadRequest` method loads the HTML file that's specified by the `HybridWebView.Uri` property. The code specifies that the file is stored in the `Content` folder of the project. Once the web page is displayed, the `invokeCSharpAction` JavaScript function will be injected into the web page.
  - When the element the renderer is attached to changes:
    - Resources are released.

**NOTE**

The `WKWebView` class is only supported in iOS 8 and later.

## Creating the Custom Renderer on Android

The following code example shows the custom renderer for the Android platform:

```

[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.Droid
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, Android.Webkit.WebView>
    {
        const string JavascriptFunction = "function invokeCSharpAction(data){jsBridge.invokeAction(data);}";
        Context _context;

        public HybridWebViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                var webView = new Android.Webkit.WebView(_context);
                webView.Settings.JavaScriptEnabled = true;
                webView.SetWebViewClient(new JavascriptWebViewClient($"javascript: {JavascriptFunction}"));
                SetNativeControl(webView);
            }
            if (e.OldElement != null)
            {
                Control.RemoveJavascriptInterface("jsBridge");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup();
            }
            if (e.NewElement != null)
            {
                Control.AddJavascriptInterface(new JSBridge(this), "jsBridge");
                Control.LoadUrl($"file:///android_asset/Content/{Element.Uri}");
            }
        }
    }
}

```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native `WebView` control, and the `invokeCSharpAction` JavaScript function is injected into the web page, after the web page has finished loading, with the `OnPageFinished` override in the `JavascriptWebViewClient` class:

```

public class JavascriptWebViewClient : WebViewClient
{
    string _javascript;

    public JavascriptWebViewClient(string javascript)
    {
        _javascript = javascript;
    }

    public override void OnPageFinished(WebView view, string url)
    {
        base.OnPageFinished(view, url);
        view.EvaluateJavascript(_javascript, null);
    }
}

```

Once the user enters their name and clicks the HTML `button` element, the `invokeCSharpAction` JavaScript function is executed. This functionality is achieved as follows:

- Provided that the `Control` property is `null`, the following operations are carried out:

- A native `WebView` instance is created, JavaScript is enabled in the control, and a `JavascriptWebViewClient` instance is set as the implementation of `WebViewClient`.
- The `SetNativeControl` method is called to assign a reference to the native `WebView` control to the `Control` property.
- Provided that the custom renderer is attached to a new Xamarin.Forms element:
  - The `WebView.AddJavascriptInterface` method injects a new `JSBridge` instance into the main frame of the `WebView`'s JavaScript context, naming it `jsBridge`. This allows methods in the `JSBridge` class to be accessed from JavaScript.
  - The `WebView.LoadUrl` method loads the HTML file that's specified by the `HybridWebView.Uri` property. The code specifies that the file is stored in the `Content` folder of the project.
  - In the `JavascriptWebViewClient` class, the `invokeCSharpAction` JavaScript function is injected into the web page once the page has finished loading.
- When the element the renderer is attached to changes:
  - Resources are released.

When the `invokeCSharpAction` JavaScript function is executed, it in turn invokes the `JSBridge.InvokeAction` method, which is shown in the following code example:

```
public class JSBridge : Java.Lang.Object
{
    readonly WeakReference<HybridWebViewRenderer> hybridWebViewRenderer;

    public JSBridge (HybridWebViewRenderer hybridRenderer)
    {
        hybridWebViewRenderer = new WeakReference <HybridWebViewRenderer> (hybridRenderer);
    }

    [JavascriptInterface]
    [Export ("invokeAction")]
    public void InvokeAction (string data)
    {
        HybridWebViewRenderer hybridRenderer;

        if (hybridWebViewRenderer != null && hybridWebViewRenderer.TryGetTarget (out hybridRenderer))
        {
            hybridRenderer.Element.InvokeAction (data);
        }
    }
}
```

The class must derive from `Java.Lang.Object`, and methods that are exposed to JavaScript must be decorated with the `[JavascriptInterface]` and `[Export]` attributes. Therefore, when the `invokeCSharpAction` JavaScript function is injected into the web page and is executed, it will call the `JSBridge.InvokeAction` method due to being decorated with the `[JavascriptInterface]` and `[Export("invokeAction")]` attributes. In turn, the `InvokeAction` method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

#### **NOTE**

Projects that use the `[Export]` attribute must include a reference to `Mono.Android.Export`, or a compiler error will result.

Note that the `JSBridge` class maintains a `WeakReference` to the `HybridWebViewRenderer` class. This is to avoid creating a circular reference between the two classes. For more information see [Weak References](#) on MSDN.

#### **Creating the Custom Renderer on UWP**

The following code example shows the custom renderer for UWP:

```

[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.UWP
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, Windows.UI.Xaml.Controls.WebView>
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data){window.external.notify(data);}";

        protected override void OnElementChanged(ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                SetNativeControl(new Windows.UI.Xaml.Controls.WebView());
            }
            if (e.OldElement != null)
            {
                Control.NavigationCompleted -= OnWebViewNavigationCompleted;
                Control.ScriptNotify -= OnWebViewScriptNotify;
            }
            if (e.NewElement != null)
            {
                Control.NavigationCompleted += OnWebViewNavigationCompleted;
                Control.ScriptNotify += OnWebViewScriptNotify;
                Control.Source = new Uri(string.Format("ms-appx-web:///Content//{0}", Element.Uri));
            }
        }

        async void OnWebViewNavigationCompleted(WebView sender, WebViewNavigationCompletedEventArgs args)
        {
            if (args.IsSuccess)
            {
                // Inject JS script
                await Control.InvokeScriptAsync("eval", new[] { JavaScriptFunction });
            }
        }

        void OnWebViewScriptNotify(object sender, NotifyEventArgs e)
        {
            Element.InvokeAction(e.Value);
        }
    }
}

```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native `WebView` control, and the `invokeCSharpAction` JavaScript function is injected into the web page, after the web page has loaded, with the `WebView.InvokeScriptAsync` method. Once the user enters their name and clicks the HTML `button` element, the `invokeCSharpAction` JavaScript function is executed, with the `OnWebViewScriptNotify` method being called after a notification is received from the web page. In turn, this method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

This functionality is achieved as follows:

- Provided that the `Control` property is `null`, the following operations are carried out:
  - The `SetNativeControl` method is called to instantiate a new native `WebView` control and assign a reference to it to the `Control` property.
- Provided that the custom renderer is attached to a new `Xamarin.Forms` element:
  - Event handlers for the `NavigationCompleted` and `ScriptNotify` events are registered. The `NavigationCompleted` event fires when either the native `WebView` control has finished loading the current content or if navigation has failed. The `ScriptNotify` event fires when the content in the native `WebView` control uses JavaScript to pass a string to the application. The web page fires the `ScriptNotify` event by

calling `window.external.notify` while passing a `string` parameter.

- The `WebView.Source` property is set to the URI of the HTML file that's specified by the `HybridWebView.Uri` property. The code assumes that the file is stored in the `Content` folder of the project. Once the web page is displayed, the `NavigationCompleted` event will fire and the `OnWebViewNavigationCompleted` method will be invoked. The `invokeCSharpAction` JavaScript function will then be injected into the web page with the `WebView.InvokeScriptAsync` method, provided that the navigation completed successfully.
- When the element the renderer is attached to changes:
  - Events are unsubscribed from.

## Summary

This article has demonstrated how to create a custom renderer for a `HybridWebView` custom control, that demonstrates how to enhance the platform-specific web controls to allow C# code to be invoked from JavaScript.

## Related Links

- [CustomRendererHybridWebView \(sample\)](#)
- [Call C# from JavaScript](#)

# Implementing a video player

6/8/2018 • 2 minutes to read • [Edit Online](#)

It is sometimes desirable to play video files in a Xamarin.Forms application. This series of articles discusses how to write custom renderers for iOS, Android, and the Universal Windows Platform (UWP) for a Xamarin.Forms class named `VideoPlayer`.

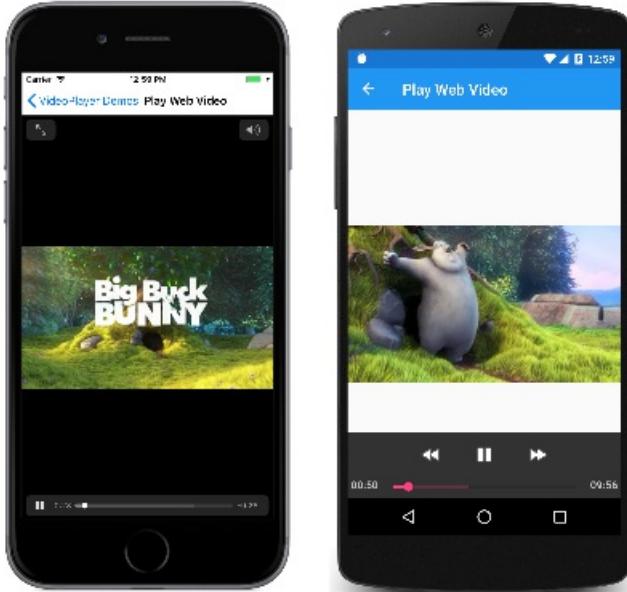
In the [VideoPlayerDemos](#) sample, all the files that implement and support `VideoPlayer` are in folders named `FormsVideoLibrary` and identified with the namespace `FormsVideoLibrary` or namespaces that begin `FormsVideoLibrary`. This organization and naming should make it easy to copy the video player files into your own Xamarin.Forms solution.

`VideoPlayer` can play video files from three types of sources:

- The Internet using a URL
- A resource embedded in the platform application
- The device's video library

Video players require *transport controls*, which are buttons for playing and pausing the video, and a positioning bar that shows the progress through the video and allows the user to skip quickly to a different location.

`VideoPlayer` can use either the transport controls and positioning bar provided by the platform (as shown below), or you can supply custom transport controls and a positioning bar. Here's the program running under iOS, Android, and the Universal Windows Platform:



Of course, you can turn the phone sideways for a larger view.

A more sophisticated video player would have some additional features, such as volume control, a mechanism to interrupt the video when a telephone call comes through, and a way of keeping the screen active during playback.

The following series of articles progressively shows how the platform renderers and supporting classes are built:

## Creating the platform video players

Each platform requires a `VideoPlayerRenderer` class that creates and maintains a video player control supported by the platform. This article shows the structure of the renderer classes, and how the players are created.

## Playing a Web video

Probably the most common source of videos for a video player is the Internet. This article describes how a Web video can be referenced and used as a source for the video player.

## Binding video sources to the player

This article uses a `ListView` to present a collection of videos to play. One program shows how the code-behind file can set the video source of the video player, but a second program shows how you can use data binding between the `ListView` and the video player.

## Loading application resource videos

Videos can be embedded as resources in the platform projects. This article shows how to store those resources and later load them into the program to be played by the video player.

## Accessing the device's video library

When a video is created using the device's camera, the video file is stored in the device's image library. This article shows how to access the device's image picker to select the video, and then play it using the video player.

## Custom video transport controls

Although the video players on each platform provide their own transport controls in the form of buttons for **Play** and **Pause**, you can suppress the display of those buttons and supply your own. This article shows you how.

## Custom video positioning

Each of the platform video players has a position bar that shows the progress of the video and allows you to skip ahead or back to a particular position. This article demonstrates how you can replace that position bar with a custom control.

## Related Links

- [Video Player Demos \(sample\)](#)

# Creating the platform video players

11/20/2018 • 8 minutes to read • [Edit Online](#)

The **VideoPlayerDemos** solution contains all the code to implement a video player for Xamarin.Forms. It also includes a series of pages that demonstrates how to use the video player within an application. All the `VideoPlayer` code and its platform renderers reside in project folders named `FormsVideoLibrary`, and also use the namespace `FormsVideoLibrary`. This should make it easy to copy the files into your own application and reference the classes.

## The video player

The `VideoPlayer` class is part of the **VideoPlayerDemos** .NET Standard library that is shared among the platforms. It derives from `View`:

```
using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
    }
}
```

The members of this class (and the `IVideoPlayerController` interface) are described in the articles that follow.

Each of the platforms contains a class named `VideoPlayerRenderer` that contains the platform-specific code to implement a video player. The primary task of this renderer is to create a video player for that platform.

### The iOS player view controller

Several classes are involved when implementing a video player in iOS. The application first creates an `AVPlayerViewController` and then sets the `Player` property to an object of type `AVPlayer`. Additional classes are required when the player is assigned a video source.

Like all renderers, the iOS `VideoPlayerRenderer` contains an `ExportRenderer` attribute that identifies the `VideoPlayer` view with the renderer:

```

using System;
using System.ComponentModel;
using System.IO;

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using UIKit;

using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
    typeof(FormsVideoLibrary.iOS.VideoPlayerRenderer))]

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
    }
}

```

Generally a renderer that sets a platform control derives from the `ViewRenderer<View, NativeView>` class, where `View` is the Xamarin.Forms `View` derivative (in this case, `VideoPlayer`) and `NativeView` is an iOS `UIView` derivative for the renderer class. For this renderer, that generic argument is simply set to `UIView`, for reasons you'll see shortly.

When a renderer is based on a `UIViewController` derivative (as this one is), then the class should override the `ViewController` property and return the view controller, in this case `AVPlayerViewController`. That is the purpose of the `_playerViewController` field:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        AVPlayerItem playerItem;
        AVPlayerViewController _playerViewController; // solely for ViewController property

        public override UIViewController ViewController => _playerViewController;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    // Create AVPlayerViewController
                    _playerViewController = new AVPlayerViewController();

                    // Set Player property to AVPlayer
                    player = new AVPlayer();
                    _playerViewController.Player = player;

                    // Use the View from the controller as the native control
                    SetNativeControl(_playerViewController.View);
                }
                ...
            }
        }
        ...
    }
}

```

The primary responsibility of the `OnElementChanged` override is to check if the `Control` property is `null` and, if so, create a platform control, and pass it to the `SetNativeControl` method. In this case, that object is only available from the `View` property of the `AVPlayerViewController`. That `UIView` derivative happens to be a private class named `AVPlayerView`, but because it's private, it cannot be explicitly specified as the second generic argument to `ViewRenderer`.

Generally the `Control` property of the renderer class thereafter refers to the `UIView` used to implement the renderer, but in this case the `Control` property is not used elsewhere.

## The Android video view

The Android renderer for `videoPlayer` is based on the Android `VideoView` class. However, if `VideoView` is used by itself to play a video in a Xamarin.Forms application, the video fills the area allotted for the `VideoPlayer` without maintaining the correct aspect ratio. For this reason (as you'll see shortly), the `VideoView` is made a child of an Android `RelativeLayout`. A `using` directive defines `ARelativeLayout` to distinguish it from the Xamarin.Forms `RelativeLayout`, and that's the second generic argument in the `ViewRenderer`:

```
using System;
using System.ComponentModel;
using System.IO;

using Android.Content;
using Android.Media;
using Android.Widget;
using ARelativeLayout = Android.Widget.RelativeLayout;

using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
    typeof(FormsVideoLibrary.Droid.VideoPlayerRenderer))]

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        public VideoPlayerRenderer(Context context) : base(context)
        {
        }
        ...
    }
}
```

Beginning in Xamarin.Forms 2.5, Android renderers should include a constructor with a `Context` argument.

The `OnElementChanged` override creates both the `VideoView` and `RelativeLayout` and sets the layout parameters for the `VideoView` to center it within the `RelativeLayout`.

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        MediaController mediaController; // Used to display transport controls
        bool isPrepared;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    // Save the VideoView for future reference
                    videoView = new VideoView(Context);

                    // Put the VideoView in a RelativeLayout
                    ARelativeLayout relativeLayout = new ARelativeLayout(Context);
                    relativeLayout.AddView(videoView);

                    // Center the VideoView in the RelativeLayout
                    ARelativeLayout.LayoutParams layoutParams =
                        new ARelativeLayout.LayoutParams(LayoutParams.MatchParent, LayoutParams.MatchParent);
                    layoutParams.AddRule(LayoutRules.CenterInParent);
                    videoView.LayoutParameters = layoutParams;

                    // Handle a VideoView event
                    videoView.Prepared += OnVideoViewPrepared;

                    // Use the RelativeLayout as the native control
                    SetNativeControl(relativeLayout);
                }
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null && videoView != null)
            {
                videoView.Prepared -= OnVideoViewPrepared;
            }
            base.Dispose(disposing);
        }

        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            isPrepared = true;
            ...
        }
        ...
    }
}

```

A handler for the `Prepared` event is attached in this method and detached in the `Dispose` method. This event is fired when the `videoView` has sufficient information to begin playing a video file.

## The UWP media element

In the Universal Windows Platform (UWP), the most common video player is `MediaElement`. That documentation of `MediaElement` indicates that the `MediaPlayerElement` should be used instead when it's only necessary to support versions of Windows 10 beginning with build 1607.

The `OnElementChanged` override needs to create a `MediaElement`, set a couple of event handlers, and pass the `MediaElement` object to `SetNativeControl`:

```
using System;
using System.ComponentModel;

using Windows.Storage;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;

using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
    typeof(FormsVideoLibrary.UWP.VideoPlayerRenderer))]

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    MediaElement mediaElement = new MediaElement();
                    SetNativeControl(mediaElement);

                    mediaElement.MediaOpened += OnMediaElementMediaOpened;
                    mediaElement.CurrentStateChanged += OnMediaElementCurrentStateChanged;
                }
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null)
            {
                Control.MediaOpened -= OnMediaElementMediaOpened;
                Control.CurrentStateChanged -= OnMediaElementCurrentStateChanged;
            }

            base.Dispose(disposing);
        }
        ...
    }
}
```

The two event handlers are detached in the `Dispose` event for the renderer.

## Showing the transport controls

All the video players included in the platforms support a default set of transport controls that include buttons for playing and pausing, and a bar to indicate the current position within the video, and to move to a new position.

The `VideoPlayer` class defines a property named `AreTransportControlsEnabled` and sets the default value to `true`:

```

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // AreTransportControlsEnabled property
        public static readonly BindableProperty AreTransportControlsEnabledProperty =
            BindableProperty.Create(nameof(AreTransportControlsEnabled), typeof(bool), typeof(VideoPlayer),
true);

        public bool AreTransportControlsEnabled
        {
            set { SetValue(AreTransportControlsEnabledProperty, value); }
            get { return (bool)GetValue(AreTransportControlsEnabledProperty); }
        }
        ...
    }
}

```

Although this property has both `set` and `get` accessors, the renderer has to handle cases only when the property is set. The `get` accessor simply returns the current value of the property.

Properties such as `AreTransportControlsEnabled` are handled in platform renderers in two ways:

- The first time is when Xamarin.Forms creates a `VideoPlayer` element. This is indicated in the `OnElementChanged` override of the renderer when the `NewElement` property is not `null`. At this time, the renderer can set its own platform video player from the property's initial value as defined in the `VideoPlayer`.
- If the property in `VideoPlayer` later changes, then the `OnElementPropertyChanged` method in the renderer is called. This allows the renderer to update the platform video player based on the new property setting.

The following sections discuss how the `AreTransportControlsEnabled` property is handled on each platform.

## iOS playback controls

The property of the iOS `AVPlayerViewController` that governs the display of transport controls is `ShowPlaybackControls`. Here's how that property is set in the iOS `VideoViewRenderer`:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        AVPlayerViewController _playerViewController;           // solely for ViewController property

        public override UIViewController ViewController => _playerViewController;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            ((AVPlayerViewController)ViewController).ShowsPlaybackControls =
Element.AreTransportControlsEnabled;
        }
        ...
    }
}

```

The `Element` property of the renderer refers to the `VideoPlayer` class.

### **The Android media controller**

In Android, displaying the transport controls requires creating a `MediaController` object and associating it with the `VideoView` object. The mechanics are demonstrated in the `SetAreTransportControlsEnabled` method:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        MediaController mediaController; // Used to display transport controls
        bool isPrepared;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            if (Element.AreTransportControlsEnabled)
            {
                mediaController = new MediaController(Context);
                mediaController.SetMediaPlayer(videoView);
                videoView.SetMediaController(mediaController);
            }
            else
            {
                videoView.SetMediaController(null);

                if (mediaController != null)
                {
                    mediaController.SetMediaPlayer(null);
                    mediaController = null;
                }
            }
        }
        ...
    }
}

```

## The UWP Transport Controls property

The UWP `MediaElement` defines a property named `AreTransportControlsEnabled`, so that property is set from the `VideoPlayer` property of the same name:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            Control.AreTransportControlsEnabled = Element.AreTransportControlsEnabled;
        }
        ...
    }
}

```

One more property is necessary to begin playing a video: This is the crucial `Source` property that references a video file. Implementing the `Source` property is described in the next article, [Playing a Web Video](#).

## Related Links

- [Video Player Demos \(sample\)](#)

# Playing a Web video

11/20/2018 • 8 minutes to read • [Edit Online](#)

The `VideoPlayer` class defines a `Source` property used to specify the source of the video file, as well as an `AutoPlay` property. `AutoPlay` has a default setting of `true`, which means that the video should begin playing automatically after `Source` has been set:

```
using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Source property
        public static readonly BindableProperty SourceProperty =
            BindableProperty.Create(nameof(Source), typeof(VideoSource), typeof(VideoPlayer), null);

        [TypeConverter(typeof(VideoSourceConverter))]
        public VideoSource Source
        {
            set { SetValue(SourceProperty, value); }
            get { return (VideoSource)GetValue(SourceProperty); }
        }

        // AutoPlay property
        public static readonly BindableProperty AutoPlayProperty =
            BindableProperty.Create(nameof(AutoPlay), typeof(bool), typeof(VideoPlayer), true);

        public bool AutoPlay
        {
            set { SetValue(AutoPlayProperty, value); }
            get { return (bool)GetValue(AutoPlayProperty); }
        }
        ...
    }
}
```

The `Source` property is of type `VideoSource`, which is patterned after the `Xamarin.Forms` `ImageSource` abstract class, and its three derivatives, `UriImageSource`, `FileImageSource`, and `StreamImageSource`. No stream option is available for the `VideoPlayer` however, because iOS and Android do not support playing a video from a stream.

## Video sources

The abstract `VideoSource` class consists solely of three static methods that instantiate the three classes that derive from `VideoSource`:

```

namespace FormsVideoLibrary
{
    [TypeConverter(typeof(VideoSourceConverter))]
    public abstract class VideoSource : Element
    {
        public static VideoSource FromUri(string uri)
        {
            return new UriVideoSource() { Uri = uri };
        }

        public static VideoSourceFromFile(string file)
        {
            return new FileVideoSource() { File = file };
        }

        public static VideoSource FromResource(string path)
        {
            return new ResourceVideoSource() { Path = path };
        }
    }
}

```

The `UriVideoSource` class is used to specify a downloadable video file with a URI. It defines a single property of type `string`:

```

namespace FormsVideoLibrary
{
    public class UriVideoSource : VideoSource
    {
        public static readonly BindableProperty UriProperty =
            BindableProperty.Create(nameof(Uri), typeof(string), typeof(UriVideoSource));

        public string Uri
        {
            set { SetValue(UriProperty, value); }
            get { return (string)GetValue(UriProperty); }
        }
    }
}

```

Handling objects of type `UriVideoSource` is described below.

The `ResourceVideoSource` class is used to access video files that are stored as embedded resources in the platform application, also specified with a `string` property:

```

namespace FormsVideoLibrary
{
    public class ResourceVideoSource : VideoSource
    {
        public static readonly BindableProperty PathProperty =
            BindableProperty.Create(nameof(Path), typeof(string), typeof(ResourceVideoSource));

        public string Path
        {
            set { SetValue(PathProperty, value); }
            get { return (string)GetValue(PathProperty); }
        }
    }
}

```

Handling objects of type `ResourceVideoSource` is described in the article [Loading Application Resource Videos](#). The

`VideoPlayer` class has no facility to load a video file stored as a resource in the .NET Standard library.

The `FileVideoSource` class is used to access video files from the device's video library. The single property is also of type `string`:

```
namespace FormsVideoLibrary
{
    public class FileVideoSource : VideoSource
    {
        public static readonly BindableProperty FileProperty =
            BindableProperty.Create(nameof(File), typeof(string), typeof(FileVideoSource));

        public string File
        {
            set { SetValue(FileProperty, value); }
            get { return (string)GetValue(FileProperty); }
        }
    }
}
```

Handling objects of type `FileVideoSource` is described in the article [Accessing the Device's Video Library](#).

The `VideoSource` class includes a `TypeConverter` attribute that references `VideoSourceConverter`:

```
namespace FormsVideoLibrary
{
    [TypeConverter(typeof(VideoSourceConverter))]
    public abstract class VideoSource : Element
    {
        ...
    }
}
```

This type converter is invoked when the `Source` property is set to a string in XAML. Here's the `VideoSourceConverter` class:

```
namespace FormsVideoLibrary
{
    public class VideoSourceConverter : TypeConverter
    {
        public override object ConvertFromInvariantString(string value)
        {
            if (!String.IsNullOrWhiteSpace(value))
            {
                Uri uri;
                return Uri.TryCreate(value, UriKind.Absolute, out uri) && uri.Scheme != "file" ?
                    VideoSource.FromUri(value) : VideoSource.FromResource(value);
            }

            throw new InvalidOperationException("Cannot convert null or whitespace to ImageSource");
        }
    }
}
```

The `ConvertFromInvariantString` method attempts to convert the string to a `Uri` object. If that succeeds, and the scheme is not `file:`, then the method returns a `UriVideoSource`. Otherwise, it returns a `ResourceVideoSource`.

## Setting the video source

All the other logic involving video sources is implemented in the individual platform renderers. The following

sections show how the platform renderers play videos when the `Source` property is set to a `UriVideoSource` object.

### The iOS video source

Two sections of the `VideoPlayerRenderer` are involved in setting the video source of the video player. When Xamarin.Forms first creates an object of type `videoPlayer`, the `OnElementChanged` method is called with the `NewElement` property of the arguments object set to that `VideoPlayer`. The `OnElementChanged` method calls `SetSource`:

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            ...
        }
        ...
    }
}
```

Later on, when the `Source` property is changed, the `OnElementPropertyChanged` method is called with a `PropertyName` property of "Source", and `SetSource` is called again.

To play a video file in iOS, an object of type `AVAsset` is first created to encapsulate the video file, and that is used to create an `AVPlayerItem`, which is then handed off to the `AVPlayer` object. Here's how the `SetSource` method handles the `Source` property when it's of type `UriVideoSource`:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        AVPlayerItem playerItem;
        ...
        void SetSource()
        {
            AVAsset asset = null;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    asset = AVAsset.FromUrl(new NSUrl(uri));
                }
            }
            ...
            if (asset != null)
            {
                playerItem = new AVPlayerItem(asset);
            }
            else
            {
                playerItem = null;
            }

            player.ReplaceCurrentItemWithPlayerItem(playerItem);

            if (playerItem != null && Element.AutoPlay)
            {
                player.Play();
            }
        }
        ...
    }
}

```

The `AutoPlay` property has no analogue in the iOS video classes, so the property is examined at the end of the `SetSource` method to call the `Play` method on the `AVPlayer` object.

In some cases, videos continued playing after the page with the `VideoPlayer` navigated back to the home page. To stop the video, the `ReplaceCurrentItemWithPlayerItem` is also set in the `Dispose` override:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void Dispose(bool disposing)
        {
            base.Dispose(disposing);

            if (player != null)
            {
                player.ReplaceCurrentItemWithPlayerItem(null);
            }
        }
        ...
    }
}

```

## The Android video source

The Android `VideoPlayerRenderer` needs to set the video source of the player when the `VideoPlayer` is first created and later when the `Source` property changes:

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                ...
            }
        }
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            ...
        }
        ...
    }
}
```

The `SetSource` method handles objects of type `UriVideoSource` by calling `SetVideoUri` on the `VideoView` with an Android `Uri` object created from the string URI. The `Uri` class is fully qualified here to distinguish it from the .NET `Uri` class:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    videoView.SetVideoURI(Android.Net.Uri.Parse(uri));
                    hasSetSource = true;
                }
            }
            ...
            if (hasSetSource && Element.AutoPlay)
            {
                videoView.Start();
            }
        }
        ...
    }
}

```

The Android `VideoView` doesn't have a corresponding `AutoPlay` property, so the `Start` method is called if a new video has been set.

There is a difference between the behavior of the iOS and Android renderers if the `Source` property of `VideoPlayer` is set to `null`, or if the `Uri` property of `UriVideoSource` is set to `null` or a blank string. If the iOS video player is currently playing a video, and `Source` is set to `null` (or the string is `null` or blank), `ReplaceCurrentItemWithPlayerItem` is called with `null` value. The current video is replaced and stops playing.

Android does not support a similar facility. If the `Source` property is set to `null`, the `SetSource` method simply ignores it, and the current video continues to play.

### The UWP video source

The UWP `MediaElement` defines an `AutoPlay` property, which is handled in the renderer like any other property:

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                SetAutoPlay();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            else if (args.PropertyName == VideoPlayer.AutoPlayProperty.PropertyName)
            {
                SetAutoPlay();
            }
            ...
        }
        ...
    }
}
```

The `SetSource` property handles a `UriVideoSource` object by setting the `Source` property of `MediaElement` to a .NET `Uri` value, or to `null` if the `Source` property of `VideoPlayer` is set to `null`:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        async void SetSource()
        {
            bool hasSetSource = false;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    Control.Source = new Uri(uri);
                    hasSetSource = true;
                }
            }
            ...
            if (!hasSetSource)
            {
                Control.Source = null;
            }
        }

        void SetAutoPlay()
        {
            Control.AutoPlay = Element.AutoPlay;
        }
        ...
    }
}

```

## Setting a URL source

With the implementation of these properties in the three renderers, it's possible to play a video from a URL source. The **Play Web Video** page in the **VideoPlayDemos** program is defined by the following XAML file:

```

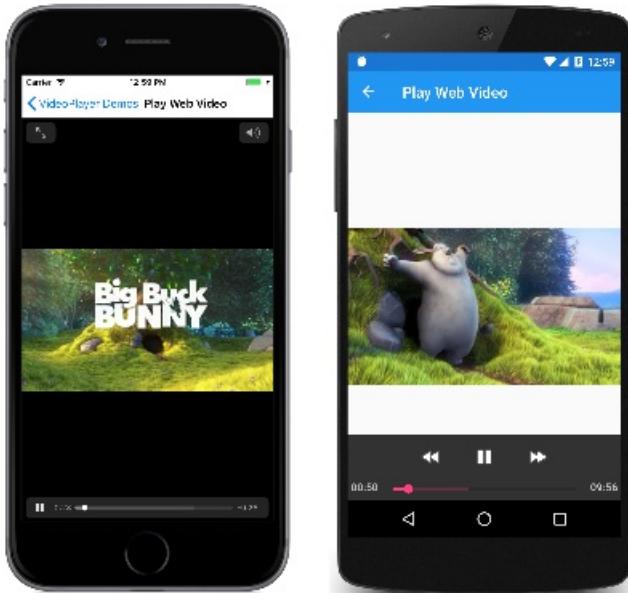
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.PlayWebVideoPage"
    Title="Play Web Video">

    <video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />

</ContentPage>

```

The `VideoSourceConverter` class converts the string to a `UriVideoSource`. When you navigate to the **Play Web Video** page, the video begins loading and starts playing when a sufficient quantity of data has been downloaded and buffered. The video is about 10 minutes in length:



On each of the platforms, the transport controls fade out if they're not used but can be restored to view by tapping the video.

You can prevent the video from automatically starting by setting the `AutoPlay` property to `false`:

```
<video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
    AutoPlay="false" />
```

You'll need to press the **Play** button to start the video.

Similarly, you can suppress the display of the transport controls by setting the `AreTransportControlsEnabled` property to `false`:

```
<video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
    AreTransportControlsEnabled="False" />
```

If you set both properties to `false`, then the video won't begin playing and there will be no way to start it! You would need to call `Play` from the code-behind file, or to create your own transport controls as described in the article [Implementing Custom Video Transport Controls](#).

The **App.xaml** file includes resources for two additional videos:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.App">
    <Application.Resources>
        <ResourceDictionary>

            <video:UriVideoSource x:Key="ElephantsDream"
                Uri="https://archive.org/download/ElephantsDream/ed_hd_512kb.mp4" />

            <video:UriVideoSource x:Key="BigBuckBunny"
                Uri="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />
        />

            <video:UriVideoSource x:Key="Sintel"
                Uri="https://archive.org/download/Sintel/sintel-2048-stereo_512kb.mp4" />
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

To reference one of these other movies, you can replace the explicit URL in the **PlayWebVideo.xaml** file with a

`StaticResource` markup extension, in which case `VideoSourceConverter` is not required to create the `UriVideoSource` object:

```
<video:VideoPlayer Source="{StaticResource ElephantsDream}" />
```

Alternatively, you can set the `Source` property from a video file in a `ListView`, as described in the next article, [Binding Video Sources to the Player](#).

## Related Links

- [Video Player Demos \(sample\)](#)

# Binding video sources to the player

6/8/2018 • 2 minutes to read • [Edit Online](#)

When the `Source` property of the `VideoPlayer` view is set to a new video file, the existing video stops playing and the new video begins. This is demonstrated by the **Select Web Video** page of the **VideoPlayerDemos** sample. The page includes a `ListView` with the titles of the three videos referenced from the **App.xaml** file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.SelectWebVideoPage"
    Title="Select Web Video">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
            Grid.Row="0" />

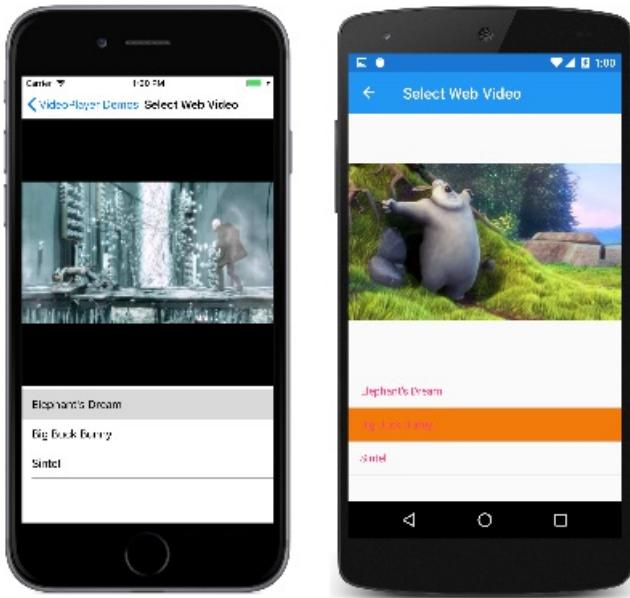
        <ListView Grid.Row="1"
            ItemSelected="OnListViewItemSelected">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type x:String}">
                    <x:String>Elephant's Dream</x:String>
                    <x:String>Big Buck Bunny</x:String>
                    <x:String>Sintel</x:String>
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </Grid>
</ContentPage>
```

When a video is selected, the `ItemSelected` event handler in the code-behind file is executed. The handler removes any blanks and apostrophes from the title and uses that as a key to obtain one of the resources defined in the **App.xaml** file. That `UriVideoSource` object is then set to the `Source` property of the `videoPlayer`.

```
namespace VideoPlayerDemos
{
    public partial class SelectWebVideoPage : ContentPage
    {
        public SelectWebVideoPage()
        {
            InitializeComponent();
        }

        void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
        {
            if (args.SelectedItem != null)
            {
                string key = ((string)args.SelectedItem).Replace(" ", "").Replace("'", "");
                videoPlayer.Source = (UriVideoSource)Application.Current.Resources[key];
            }
        }
    }
}
```

When the page first loads, no item is selected in the `ListView`, so you must select one for the video to begin playing:



The `Source` property of `videoPlayer` is backed by a bindable property, which means that it can be the target of a data binding. This is demonstrated by the **Bind to VideoPlayer** page. The markup in the **BindToVideoPlayer.xaml** file is supported by the following class that encapsulates a title of a video and a corresponding `videoSource` object:

```
namespace VideoPlayerDemos
{
    public class VideoInfo
    {
        public string DisplayName { set; get; }

        public Uri Uri { get; set; }

        public override string ToString()
        {
            return DisplayName;
        }
    }
}
```

The `ListView` in the **BindToVideoPlayer.xaml** file contains an array of these `VideoInfo` objects, each one initialized with a video title and the `Uri` object from the resource dictionary in **App.xaml**:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:VideoPlayerDemos"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.BindToVideoPlayerPage"
    Title="Bind to VideoPlayer">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
            Grid.Row="0"
            Source="{Binding Source={x:Reference listView},
                Path=SelectedItem.VideoSource}" />

        <ListView x:Name="listView"
            Grid.Row="1">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:VideoInfo}">
                    <local:VideoInfo DisplayName="Elephant's Dream"
                        VideoSource="{StaticResource ElephantsDream}" />

                    <local:VideoInfo DisplayName="Big Buck Bunny"
                        VideoSource="{StaticResource BigBuckBunny}" />

                    <local:VideoInfo DisplayName="Sintel"
                        VideoSource="{StaticResource Sintel}" />
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </Grid>
</ContentPage>

```

The `Source` property of the `VideoPlayer` is bound to the `ListView`. The `Path` of the binding is specified as `SelectedItem.VideoSource`, which is a compound path consisting of two properties: `SelectedItem` is a property of `ListView`. The selected item is of type `VideoInfo`, which has a `VideoSource` property.

As with the first **Select Web Video** page, no item is initially selected from the `ListView`, so you need to select one of the videos before it begins playing.

## Related Links

- [Video Player Demos \(sample\)](#)

# Loading application resource videos

11/20/2018 • 3 minutes to read • [Edit Online](#)

The custom renderers for the `VideoPlayer` view are capable of playing video files that have been embedded in the individual platform projects as application resources. However, the current version of `VideoPlayer` cannot access resources embedded in a .NET Standard library.

To load these resources, create an instance of `ResourceVideoSource` by setting the `Path` property to the filename (or the folder and filename) of the resource. Alternatively, you can call the static `VideoSource.FromResource` method to reference the resource. Then, set the `ResourceVideoSource` object to the `Source` property of `VideoPlayer`.

## Storing the video files

Storing a video file in the platform project is different for each platform.

### iOS video resources

In an iOS project, you can store a video in the **Resources** folder, or a subfolder of the **Resources** folder. The video file must have a `Build Action` of `BundleResource`. Set the `Path` property of `ResourceVideoSource` to the filename, for example, **MyFile.mp4** for a file in the **Resources** folder, or **MyFolder/MyFile.mp4**, where **MyFolder** is a subfolder of **Resources**.

In the **VideoPlayerDemos** solution, the **VideoPlayerDemos.iOS** project contains a subfolder of **Resources** named **Videos** containing a file named **iOSApiVideo.mp4**. This is a short video that shows you how to use the Xamarin web site to find documentation for the iOS `AVPlayerViewController` class.

### Android video resources

In an Android project, videos must be stored in a subfolder of **Resources** named **raw**. The **raw** folder cannot contain subfolders. Give the video file a `Build Action` of `AndroidResource`. Set the `Path` property of `ResourceVideoSource` to the filename, for example, **MyFile.mp4**.

The **VideoPlayerDemos.Android** project contains a subfolder of **Resources** named **raw**, which contains a file named **AndroidApiVideo.mp4**.

### UWP video resources

In a Universal Windows Platform project, you can store videos in any folder in the project. Give the file a `Build Action` of `Content`. Set the `Path` property of `ResourceVideoSource` to the folder and filename, for example, **MyFolder/MyVideo.mp4**.

The **VideoPlayerDemos.UWP** project contains a folder named **Videos** with the file **UWPApiVideo.mp4**.

## Loading the video files

Each of the platform renderer classes contains code in its `SetSource` method for loading video files stored as resources.

### iOS resource loading

The iOS version of `VideoPlayerRenderer` uses the `GetUrlForResource` method of `NSBundle` for loading the resource. The complete path must be divided into a filename, extension, and directory. The code uses the `Path` class in the .NET `System.IO` namespace for dividing the file path into these components:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void SetSource()
        {
            AVAsset asset = null;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string path = (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhiteSpace(path))
                {
                    string directory = Path.GetDirectoryName(path);
                    string filename = Path.GetFileNameWithoutExtension(path);
                    string extension = Path.GetExtension(path).Substring(1);
                    NSUrl url = NSBundle.MainBundle.GetUrlForResource(filename, extension, directory);
                    asset = AVAsset.FromUrl(url);
                }
            }
            ...
        }
        ...
    }
}

```

## Android resource loading

The Android `VideoPlayerRenderer` uses the filename and package name to construct a `Uri` object. The package name is the name of the application, in this case **VideoPlayerDemos.Android**, which can be obtained from the static `Context.PackageName` property. The resultant `Uri` object is then passed to the `SetVideoURI` method of `VideoView`:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string package = Context.PackageName;
                string path = (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhiteSpace(path))
                {
                    string filename = Path.GetFileNameWithoutExtension(path).ToLowerInvariant();
                    string uri = "android.resource://" + package + "/raw/" + filename;
                    videoView.SetVideoURI(Android.Net.Uri.Parse(uri));
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}

```

## UWP resource loading

The UWP `VideoPlayerRenderer` constructs a `Uri` object for the path and sets it to the `Source` property of `MediaElement`:

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        async void SetSource()
        {
            bool hasSetSource = false;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string path = "ms-appx:/// " + (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhiteSpace(path))
                {
                    Control.Source = new Uri(path);
                    hasSetSource = true;
                }
            }
            ...
        }
    }
}
```

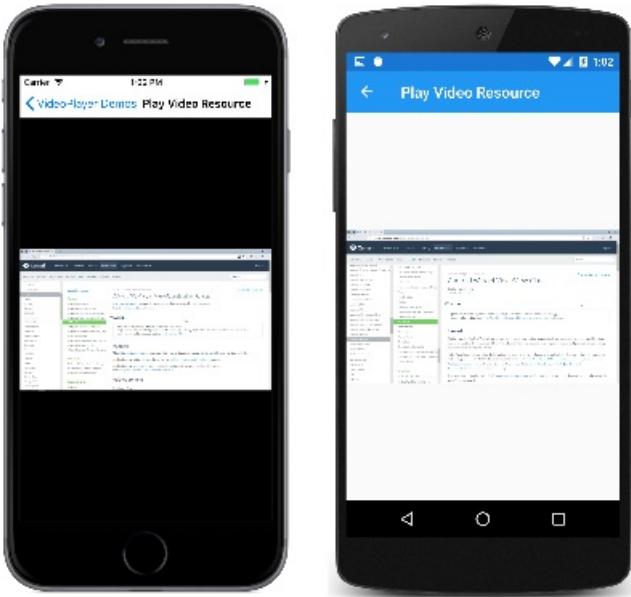
## Playing the resource file

The **Play Video Resource** page in the **VideoPlayerDemos** solution uses the `OnPlatform` class to specify the video file for each platform:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.PlayVideoResourcePage"
    Title="Play Video Resource">
    <video:VideoPlayer>
        <video:VideoPlayer.Source>
            <video:ResourceVideoSource>
                <video:ResourceVideoSource.Path>
                    <OnPlatform x:TypeArguments="x:String">
                        <On Platform="iOS" Value="Videos/iOSApiVideo.mp4" />
                        <On Platform="Android" Value="AndroidApiVideo.mp4" />
                        <On Platform="UWP" Value="Videos/UWPApiVideo.mp4" />
                    </OnPlatform>
                </video:ResourceVideoSource.Path>
            </video:ResourceVideoSource>
        </video:VideoPlayer.Source>
    </video:VideoPlayer>
</ContentPage>
```

If the iOS resource is stored in the **Resources** folder, and if the UWP resource is stored in the root folder of the project, you can use the same filename for each platform. If that is the case, then you can set that name directly to the `Source` property of `VideoPlayer`.

Here's that page running:



You've now seen how to [load videos from a Web URI](#) and how to play embedded resources. In addition, you can [load videos from the device's video library](#).

## Related Links

- [Video Player Demos \(sample\)](#)

# Accessing the device's video library

11/20/2018 • 6 minutes to read • [Edit Online](#)

Most modern mobile devices and desktop computers have the ability to record videos using the device's camera. The videos that a user creates are then stored as files on the device. These files can be retrieved from the image library and played by the `VideoPlayer` class just like any other video.

## The photo picker dependency service

Each of the platforms includes a facility that allows the user to select a photo or video from the device's image library. The first step in playing a video from the device's image library is building a dependency service that invokes the image picker on each platform. The dependency service described below is very similar to one defined in the [Picking a Photo from the Picture Library](#) article, except that the video picker returns a filename rather than a `Stream` object.

The .NET Standard library project defines an interface named `IVideoPicker` for the dependency service:

```
namespace FormsVideoLibrary
{
    public interface IVideoPicker
    {
        Task<string> GetVideoFileAsync();
    }
}
```

Each of the platforms contains a class named `VideoPicker` that implements this interface.

### The iOS video picker

The iOS `VideoPicker` uses the iOS `UIImagePickerController` to access the image library, specifying that it should be restricted to videos (referred to as "movies") in the iOS `MediaType` property. Notice that `VideoPicker` explicitly implements the `IVideoPicker` interface. Notice also the `Dependency` attribute that identifies this class as a dependency service. These are the two requirements that allow Xamarin.Forms to find the dependency service in the platform project:

```

using System;
using System.Threading.Tasks;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(FormsVideoLibrary.iOS.VideoPicker))]

namespace FormsVideoLibrary.iOS
{
    public class VideoPicker : IVideoPicker
    {
        TaskCompletionSource<string> taskCompletionSource;
        UIImagePickerController imagePicker;

        public Task<string> GetVideoFileAsync()
        {
            // Create and define UIImagePickerController
            imagePicker = new UIImagePickerController
            {
                SourceType = UIImagePickerControllerSourceType.SavedPhotosAlbum,
                MediaTypes = new string[] { "public.movie" }
            };

            // Set event handlers
            imagePicker.FinishedPickingMedia += OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled += OnImagePickerCancelled;

            // Present UIImagePickerController;
            UIWindow window = UIApplication.SharedApplication.KeyWindow;
            var viewController = window.RootViewController;
            viewController.PresentModalViewController(imagePicker, true);

            // Return Task object
            taskCompletionSource = new TaskCompletionSource<string>();
            return taskCompletionSource.Task;
        }

        void OnImagePickerFinishedPickingMedia(object sender, UIImagePickerControllerMediaPickedEventArgs args)
        {
            if (args.MediaType == "public.movie")
            {
                taskCompletionSource.SetResult(args.MediaUrl.AbsoluteString);
            }
            else
            {
                taskCompletionSource.SetResult(null);
            }
            imagePicker.DismissModalViewController(true);
        }

        void OnImagePickerCancelled(object sender, EventArgs args)
        {
            taskCompletionSource.SetResult(null);
            imagePicker.DismissModalViewController(true);
        }
    }
}

```

## The Android video picker

The Android implementation of `IVideoPicker` requires a callback method that is part of the application's activity. For that reason, the `MainActivity` class defines two properties, a field, and a callback method:

```

namespace VideoPlayerDemos.Droid
{
    ...
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            Current = this;
            ...
        }

        // Field, properties, and method for Video Picker
        public static MainActivity Current { private set; get; }

        public static readonly int PickImageId = 1000;

        public TaskCompletionSource<string> PickImageTaskCompletionSource { set; get; }

        protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
        {
            base.OnActivityResult(requestCode, resultCode, data);

            if (requestCode == PickImageId)
            {
                if ((resultCode == Result.Ok) && (data != null))
                {
                    // Set the filename as the completion of the Task
                    PickImageTaskCompletionSource.SetResult(data.DataString);
                }
                else
                {
                    PickImageTaskCompletionSource.SetResult(null);
                }
            }
        }
    }
}

```

The `OnCreate` method in `MainActivity` stores its own instance in the static `Current` property. This allows the implementation of `IVideoPicker` to obtain the `MainActivity` instance for starting the **Select Video** chooser:

```

using System;
using System.Threading.Tasks;
using Android.Content;
using Xamarin.Forms;

// Need application's MainActivity
using VideoPlayerDemos.Droid;

[assembly: Dependency(typeof(FormsVideoLibrary.Droid.VideoPicker))]

namespace FormsVideoLibrary.Droid
{
    public class VideoPicker : IVideoPicker
    {
        public Task<string> GetVideoFileAsync()
        {
            // Define the Intent for getting images
            Intent intent = new Intent();
            intent.SetType("video/*");
            intent.SetAction(Intent.ActionGetContent);

            // Get the MainActivity instance
            MainActivity activity = MainActivity.Current;

            // Start the picture-picker activity (resumes in MainActivity.cs)
            activity.StartActivityForResult(
                Intent.CreateChooser(intent, "Select Video"),
                MainActivity.PickImageId);

            // Save the TaskCompletionSource object as a MainActivity property
            activity.PickImageTaskCompletionSource = new TaskCompletionSource<string>();

            // Return Task object
            return activity.PickImageTaskCompletionSource.Task;
        }
    }
}

```

The additions to the `MainActivity` object are the only code in the **VideoPlayerDemos** solution where normal application code needs to be altered to support the `FormsVideoLibrary` classes.

### The UWP video picker

The UWP implementation of the `IvideoPicker` interface uses the UWP `FileOpenPicker`. It begins the file search with the pictures library, and restricts the file types to MP4 and WMV (Windows Media Video):

```

using System;
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Pickers;
using Xamarin.Forms;

[assembly: Dependency(typeof(FormsVideoLibrary.UWP.VideoPicker))]

namespace FormsVideoLibrary.UWP
{
    public class VideoPicker : IVideoPicker
    {
        public async Task<string> GetVideoFileAsync()
        {
            // Create and initialize the FileOpenPicker
            FileOpenPicker openPicker = new FileOpenPicker
            {
                ViewMode = PickerViewMode.Thumbnail,
                SuggestedStartLocation = PickerLocationId.PicturesLibrary
            };

            openPicker.FileTypeFilter.Add(".wmv");
            openPicker.FileTypeFilter.Add(".mp4");

            // Get a file and return the path
            StorageFile storageFile = await openPicker.PickSingleFileAsync();
            return storageFile?.Path;
        }
    }
}

```

## Invoking the dependency service

The **Play Library Video** page of the [VideoPlayerDemos](#) program demonstrates how to use the video picker dependency service. The XAML file contains a `VideoPlayer` instance and a `Button` labeled **Show Video Library**:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.PlayLibraryVideoPage"
    Title="Play Library Video">

    <StackLayout>
        <video:VideoPlayer x:Name="videoPlayer"
            VerticalOptions="FillAndExpand" />

        <Button Text="Show Video Library"
            Margin="10"
            HorizontalOptions="Center"
            Clicked="OnShowVideoLibraryClicked" />
    </StackLayout>
</ContentPage>

```

The code-behind file contains the `Clicked` handler for the `Button`. Invoking the dependency service requires a call to `DependencyService.Get` to obtain the implementation of an `IVideoPicker` interface in the platform project. The `GetVideoFileAsync` method is then called on that instance:

```

namespace VideoPlayerDemos
{
    public partial class PlayLibraryVideoPage : ContentPage
    {
        public PlayLibraryVideoPage()
        {
            InitializeComponent();
        }

        async void OnShowVideoLibraryClicked(object sender, EventArgs args)
        {
            Button btn = (Button)sender;
            btn.IsEnabled = false;

            string filename = await DependencyService.Get<IVideoPicker>().GetVideoFileAsync();

            if (!String.IsNullOrWhiteSpace(filename))
            {
                videoPlayer.Source = new FileVideoSource
                {
                    File = filename
                };
            }

            btn.IsEnabled = true;
        }
    }
}

```

The `clicked` handler then uses that filename to create a `FileVideoSource` object and to set it to the `Source` property of the `VideoPlayer`.

Each of the `VideoPlayerRenderer` classes contains code in its `SetSource` method for objects of type `FileVideoSource`. These are shown below:

### Handling iOS files

The iOS version of `VideoPlayerRenderer` processes `FileVideoSource` objects by using the static `Asset.FromUrl` method with the filename. This creates an `AVAsset` object representing the file in the device's image library:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void SetSource()
        {
            AVAsset asset = null;
            ...
            else if (Element.Source is FileVideoSource)
            {
                string uri = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    asset = AVAsset.FromUrl(new NSUrl(uri));
                }
            }
            ...
        }
        ...
    }
}

```

## Handling Android files

When processing objects of type `FileVideoSource`, the Android implementation of `VideoPlayerRenderer` uses the `SetVideoPath` method of `VideoView` to specify the file in the device's image library:

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;
            ...
            else if (Element.Source is FileVideoSource)
            {
                string filename = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(filename))
                {
                    videoView.SetVideoPath(filename);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}
```

## Handling UWP files

When handling objects of type `FileVideoSource`, the UWP implementation of the `SetSource` method needs to create a `StorageFile` object, open that file for reading, and pass the stream object to the `SetSource` method of the `MediaElement`:

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
        }

        async void SetSource()
        {
            bool hasSetSource = false;
            ...

            else if (Element.Source is FileVideoSource)
            {
                // Code requires Pictures Library in Package.appxmanifest Capabilities to be enabled
                string filename = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(filename))
                {
                    StorageFile storageFile = await StorageFile.GetFileFromPathAsync(filename);
                    IRandomAccessStreamWithContentType stream = await storageFile.OpenReadAsync();
                    Control.SetSource(stream, storageFile.ContentType);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}
```

For each platform, the video begins playing almost immediately after the video source is set because the file is on the device and doesn't need to be downloaded.

## Related Links

- [Video Player Demos \(sample\)](#)
- [Picking a Photo from the Picture Library](#)

# Custom video transport controls

11/20/2018 • 10 minutes to read • [Edit Online](#)

The transport controls of a video player include the buttons that perform the functions **Play**, **Pause**, and **Stop**. These buttons are generally identified with familiar icons rather than text, and the **Play** and **Pause** functions are generally combined into one button.

By default, the `videoPlayer` displays transport controls supported by each platform. When you set the `AreTransportControlsEnabled` property to `false`, these controls are suppressed. You can then control the `VideoPlayer` programmatically or supply your own transport controls.

## The Play, Pause, and Stop methods

The `VideoPlayer` class defines three methods named `Play`, `Pause`, and `Stop` that are implemented by firing events:

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        public event EventHandler PlayRequested;

        public void Play()
        {
            PlayRequested?.Invoke(this, EventArgs.Empty);
        }

        public event EventHandler PauseRequested;

        public void Pause()
        {
            PauseRequested?.Invoke(this, EventArgs.Empty);
        }

        public event EventHandler StopRequested;

        public void Stop()
        {
            StopRequested?.Invoke(this, EventArgs.Empty);
        }
    }
}
```

Event handlers for these events are set by the `VideoPlayerRenderer` class in each platform, as shown below:

### iOS transport implementations

The iOS version of `VideoPlayerRenderer` uses the `OnElementChanged` method to set handlers for these three events when the `NewElement` property is not `null` and detaches the event handlers when `OldElement` is not `null`:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }
        ...
        // Event handlers to implement methods
        void OnPlayRequested(object sender, EventArgs args)
        {
            player.Play();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            player.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            player.Pause();
            player.Seek(new CMTIME(0, 1));
        }
    }
}

```

The event handlers are implemented by calling methods on the `AVPlayer` object. There is no `Stop` method for `AVPlayer`, so it's simulated by pausing the video and moving the position to the beginning.

### Android transport implementations

The Android implementation is similar to the iOS implementation. The handlers for the three functions are set when `NewElement` is not `null` and detached when `OldElement` is not `null`:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }
        ...
        void OnPlayRequested(object sender, EventArgs args)
        {
            videoView.Start();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            videoView.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            videoView.StopPlayback();
        }
    }
}

```

The three functions call methods defined by `videoView`.

### **UWP transport implementations**

The UWP implementation of the three transport functions is very similar to both the iOS and Android implementations:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }

        ...
        // Event handlers to implement methods
        void OnPlayRequested(object sender, EventArgs args)
        {
            Control.Play();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            Control.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            Control.Stop();
        }
    }
}

```

## The video player status

Implementing the **Play**, **Pause**, and **Stop** functions is not sufficient for supporting transport controls. Often the **Play** and **Pause** commands are implemented with the same button that changes its appearance to indicate whether the video is currently playing or paused. Moreover, the button shouldn't even be enabled if the video has not yet loaded.

These requirements imply that the video player needs to make available a current status indicating if it's playing or paused, or if it's not yet ready to play a video. (Each platform also supports properties that indicate if the video can be paused, or can be moved to a new position, but these properties are applicable for streaming video rather than video files, so they are not supported in the `VideoPlayer` described here.)

The **VideoPlayerDemos** project includes a `VideoStatus` enumeration with three members:

```

namespace FormsVideoLibrary
{
    public enum VideoStatus
    {
        NotReady,
        Playing,
        Paused
    }
}

```

The `VideoPlayer` class defines a real-only bindable property named `Status` of type `VideoStatus`. This property is defined as read-only because it should only be set from the platform renderer:

```

using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Status read-only property
        private static readonly BindablePropertyKey StatusPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Status), typeof(VideoStatus), typeof(VideoPlayer),
            VideoStatus.NotReady);

        public static readonly BindableProperty StatusProperty = StatusPropertyKey.BindableProperty;

        public VideoStatus Status
        {
            get { return (VideoStatus)GetValue(StatusProperty); }
        }

        VideoStatus IVideoPlayerController.Status
        {
            set { SetValue(StatusPropertyKey, value); }
            get { return Status; }
        }
        ...
    }
}

```

Usually, a read-only bindable property would have a private `set` accessor on the `Status` property to allow it to be set from within the class. For a `View` derivative supported by renderers, however, the property must be set from outside the class, but only by the platform renderer.

For this reason, another property is defined with the name `IVideoPlayerController.Status`. This is an explicit interface implementation, and is made possible by the `IVideoPlayerController` interface that the `VideoPlayer` class implements:

```

namespace FormsVideoLibrary
{
    public interface IVideoPlayerController
    {
        VideoStatus Status { set; get; }

        TimeSpan Duration { set; get; }
    }
}

```

This is similar to how the `WebView` control uses the `IWebViewController` interface to implement the `CanGoBack` and

`CanGoForward` properties. (See the source code of `WebView` and its renderers for details.)

This makes it possible for a class external to `VideoPlayer` to set the `Status` property by referencing the `IVideoPlayerController` interface. (You'll see the code shortly.) The property can be set from other classes as well, but it's unlikely to be set inadvertently. Most importantly, the `Status` property cannot be set through a data binding.

To assist the renderers in keeping this `Status` property updated, the `VideoPlayer` class defines an `UpdateStatus` event that is triggered every tenth of a second:

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        public event EventHandler UpdateStatus;

        public VideoPlayer()
        {
            Device.StartTimer(TimeSpan.FromMilliseconds(100), () =>
            {
                UpdateStatus?.Invoke(this, EventArgs.Empty);
                return true;
            });
        }
        ...
    }
}
```

### The iOS status setting

The iOS `VideoPlayerRenderer` sets a handler for the `UpdateStatus` event (and detaches that handler when the underlying `VideoPlayer` element is absent), and uses the handler to set the `Status` property:

```

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.UpdateStatus += OnUpdateStatus;
                ...
            }

            if (args.OldElement != null)
            {
                args.OldElement.UpdateStatus -= OnUpdateStatus;
                ...
            }
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            VideoStatus videoStatus = VideoStatus.NotReady;

            switch (player.Status)
            {
                case AVPlayerStatus.ReadyToPlay:
                    switch (player.TimeControlStatus)
                    {
                        case AVPlayerTimeControlStatus.Playing:
                            videoStatus = VideoStatus.Playing;
                            break;

                        case AVPlayerTimeControlStatus.Paused:
                            videoStatus = VideoStatus.Paused;
                            break;
                    }
                    break;
            }

            ((IVideoPlayerController)Element).Status = videoStatus;
            ...
        }
        ...
    }
}

```

Two properties of `AVPlayer` must be accessed: The `Status` property of type `AVPlayerStatus` and the `TimeControlStatus` property of type `AVPlayerTimeControlStatus`. Notice that the `Element` property (which is the `VideoPlayer`) must be cast to `IVideoPlayerController` to set the `Status` property.

### The Android status setting

The `IsPlaying` property of the Android `VideoView` is a Boolean that only indicates if the video is playing or paused. To determine if the `VideoView` can neither play nor pause the video yet, the `Prepared` event of `VideoView` must be handled. These two handlers are set in the `OnElementChanged` method, and detached during the `Dispose` override:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        bool isPrepared;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    ...
                    videoView.Prepared += OnVideoViewPrepared;
                    ...
                }
                ...
                args.NewElement.UpdateStatus += OnUpdateStatus;
                ...
            }

            if (args.OldElement != null)
            {
                args.OldElement.UpdateStatus -= OnUpdateStatus;
                ...
            }
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null && videoView != null)
            {
                videoView.Prepared -= OnVideoViewPrepared;
            }
            if (Element != null)
            {
                Element.UpdateStatus -= OnUpdateStatus;
            }

            base.Dispose(disposing);
        }
        ...
    }
}

```

The `UpdateStatus` handler uses the `isPrepared` field (set in the `Prepared` handler) and the `IsPlaying` property to set the `Status` property:

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        bool isPrepared;
        ...
        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            isPrepared = true;
            ...
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            VideoStatus status = VideoStatus.NotReady;

            if (isPrepared)
            {
                status = videoView.isPlaying ? VideoStatus.Playing : VideoStatus.Paused;
            }
            ...
        }
        ...
    }
}
```

### The UWP status setting

The UWP `VideoPlayerRenderer` makes use of the `UpdateStatus` event, but it does not need it for setting the `Status` property. The `MediaElement` defines a `CurrentStateChanged` event that allows the renderer to be notified when the `CurrentState` property has changed. The property is detached in the `Dispose` override:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    ...
                    mediaElement.CurrentStateChanged += OnMediaElementCurrentStateChanged;
                };
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null)
            {
                ...
                Control.CurrentStateChanged -= OnMediaElementCurrentStateChanged;
            }

            base.Dispose(disposing);
        }
        ...
    }
}

```

The `CurrentState` property is of type `MediaElementState`, and maps easily into `VideoStatus`:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        void OnMediaElementCurrentStateChanged(object sender, RoutedEventArgs args)
        {
            VideoStatus videoStatus = VideoStatus.NotReady;

            switch (Control.CurrentState)
            {
                case MediaElementState.Playing:
                    videoStatus = VideoStatus.Playing;
                    break;

                case MediaElementState.Paused:
                case MediaElementState.Stopped:
                    videoStatus = VideoStatus.Paused;
                    break;
            }

            ((IVideoPlayerController)Element).Status = videoStatus;
        }
        ...
    }
}

```

# Play, Pause, and Stop Buttons

Using Unicode characters for symbolic **Play**, **Pause**, and **Stop** images is problematic. The [Miscellaneous Technical](#) section of the Unicode standard defines three symbol characters seemingly appropriate for this purpose. These are:

- 0x23F5 (black medium right-pointing triangle) or ▶ for **Play**
- 0x23F8 (double vertical bar) or □ for **Pause**
- 0x23F9 (black square) or ▨ for **Stop**

Regardless how these symbols appear in your browser (and different browsers handle them in different ways), they are not displayed consistently on the platforms supported by Xamarin.Forms. On iOS and UWP devices, the **Pause** and **Stop** characters have a graphical appearance, with a blue 3D background and a white foreground. This isn't the case on Android, where the symbol is simply blue. However, the 0x23F5 codepoint for **Play** does not have that same appearance on the UWP, and it's not even supported on iOS and Android.

For that reason, the 0x23F5 codepoint can't be used for **Play**. A good substitute is:

- 0x25B6 (black right-pointing triangle) or ► for **Play**

This is supported by each platform except that it's a plain black triangle that does not resemble the 3D appearance of **Pause** and **Stop**. One possibility is to follow the 0x25B6 codepoint with a variant code:

- 0x25B6 followed by 0xFE0F (variant 16) or ►□ for **Play**

This is what's used in the markup shown below. On iOS, it gives the **Play** symbol the same 3D appearance as the **Pause** and **Stop** buttons, but the variant doesn't work on Android and the UWP.

The [Custom Transport](#) page sets the **AreTransportControlsEnabled** property to **false** and includes an `ActivityIndicator` displayed when the video is loading, and two buttons. `DataTrigger` objects are used to enable and disable the `ActivityIndicator` and the buttons, and to switch the first button between **Play** and **Pause**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.CustomTransportPage"
    Title="Custom Transport">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
            Grid.Row="0"
            AutoPlay="False"
            AreTransportControlsEnabled="False"
            Source="{StaticResource BigBuckBunny}" />

        <ActivityIndicator Grid.Row="0"
            Color="Gray"
            IsVisible="False">
            <ActivityIndicator.Triggers>
                <DataTrigger TargetType="ActivityIndicator"
                    Binding="{Binding Source={x:Reference videoPlayer},
                        Path=Status}"
                    Value="{x:Static video:VideoStatus.NotReady}">
                    <Setter Property="IsVisible" Value="True" />
                    <Setter Property="IsRunning" Value="True" />
                </DataTrigger>
            </ActivityIndicator.Triggers>
        </ActivityIndicator>
    </Grid>
</ContentPage>
```

```

<StackLayout Grid.Row="1"
            Orientation="Horizontal"
            Margin="0, 10"
            BindingContext="{x:Reference videoPlayer}>

    <Button Text=" Play"
           HorizontalOptions="CenterAndExpand"
           Clicked="OnPlayPauseButtonClicked">
        <Button.Triggers>
            <DataTrigger TargetType="Button"
                         Binding="{Binding Status}"
                         Value="{x:Static video:VideoStatus.Playing}">
                <Setter Property="Text" Value=" Pause" />
            </DataTrigger>
        </Button.Triggers>
    </Button>

    <Button Text=" Stop"
           HorizontalOptions="CenterAndExpand"
           Clicked="OnStopButtonClicked">
        <Button.Triggers>
            <DataTrigger TargetType="Button"
                         Binding="{Binding Status}"
                         Value="{x:Static video:VideoStatus.NotReady}">
                <Setter Property="IsEnabled" Value="False" />
            </DataTrigger>
        </Button.Triggers>
    </Button>
</StackLayout>
</Grid>
</ContentPage>

```

Data triggers are described in detail in the article [Data Triggers](#).

The code-behind file has the handlers for the button `Clicked` events:

```

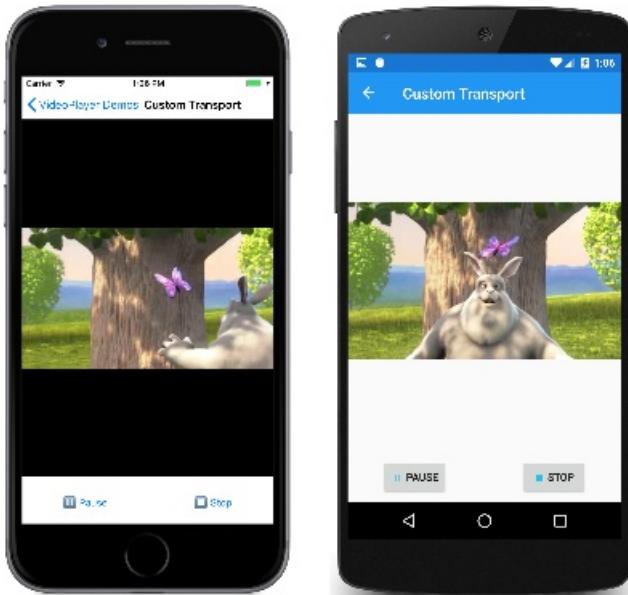
namespace VideoPlayerDemos
{
    public partial class CustomTransportPage : ContentPage
    {
        public CustomTransportPage()
        {
            InitializeComponent();
        }

        void OnPlayPauseButtonClicked(object sender, EventArgs args)
        {
            if (videoPlayer.Status == VideoStatus.Playing)
            {
                videoPlayer.Pause();
            }
            else if (videoPlayer.Status == VideoStatus.Paused)
            {
                videoPlayer.Play();
            }
        }

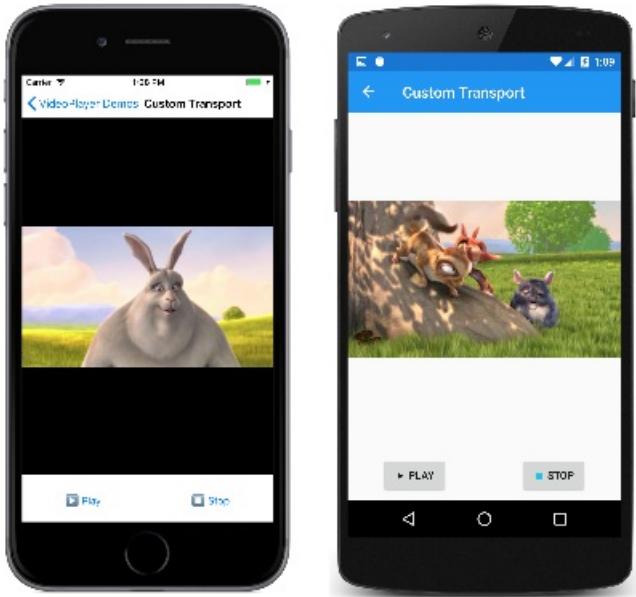
        void OnStopButtonClicked(object sender, EventArgs args)
        {
            videoPlayer.Stop();
        }
    }
}

```

Because `AutoPlay` is set to `false` in the **CustomTransport.xaml** file, you'll need to press the **Play** button when it becomes enabled to begin the video. The buttons are defined so that the Unicode characters discussed above are accompanied by their text equivalents. The buttons have a consistent appearance on each platform when the video is playing:



But on Android and UWP, the **Play** button looks very different when the video is paused:



In a production application, you'll probably want to use your own bitmap images for the buttons to achieve visual uniformity.

## Related Links

- [Video Player Demos \(sample\)](#)

# Custom video positioning

6/8/2018 • 10 minutes to read • [Edit Online](#)

The transport controls implemented by each platform include a position bar. This bar resembles a slider or scroll bar and shows the current location of the video within its total duration. In addition, the user can manipulate the position bar to move forwards or backwards to a new position in the video.

This article shows how you can implement your own custom position bar.

## The Duration property

One item of information that `VideoPlayer` needs to support a custom position bar is the duration of the video. The `VideoPlayer` defines a read-only `Duration` property of type `TimeSpan`:

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Duration read-only property
        private static readonly BindablePropertyKey DurationPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Duration), typeof(TimeSpan), typeof(VideoPlayer), new
        TimeSpan(),
            PropertyChanged: (bindable, oldValue, newValue) => ((VideoPlayer)bindable).SetTimeToEnd());

        public static readonly BindableProperty DurationProperty = DurationPropertyKey.BindableProperty;

        public TimeSpan Duration
        {
            get { return (TimeSpan)GetValue(DurationProperty); }
        }

        TimeSpan IVideoPlayerController.Duration
        {
            set { SetValue(DurationPropertyKey, value); }
            get { return Duration; }
        }
        ...
    }
}
```

Like the `Status` property described in the [previous article](#), this `Duration` property is read-only. It's defined with a private `BindablePropertyKey` and can only be set by referencing the `IVideoPlayerController` interface, which includes this `Duration` property:

```
namespace FormsVideoLibrary
{
    public interface IVideoPlayerController
    {
        VideoStatus Status { set; get; }

        TimeSpan Duration { set; get; }
    }
}
```

Also notice the property-changed handler that calls a method named `SetTimeToEnd` that is described later in this

article.

The duration of a video is *not* available immediately after the `Source` property of `VideoPlayer` is set. The video file must be partially downloaded before the underlying video player can determine its duration.

Here's how each of the platform renderers obtains the video's duration:

### Video duration in iOS

In iOS, the duration of a video is obtained from the `Duration` property of `AVPlayerItem`, but not immediately after the `AVPlayerItem` is created. It's possible to set an iOS observer for the `Duration` property, but the `VideoPlayerRenderer` obtains the duration in the `UpdateStatus` method, which is called 10 times a second:

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            if (playerItem != null)
            {
                ((IVideoPlayerController)Element).Duration = ConvertTime(playerItem.Duration);
                ...
            }
        }

        TimeSpan ConvertTime(CMTime cmTime)
        {
            return TimeSpan.FromSeconds(Double.IsNaN(cmTime.Seconds) ? 0 : cmTime.Seconds);
        }
        ...
    }
}
```

The `ConvertTime` method converts a `CMTime` object to a `TimeSpan` value.

### Video duration in Android

The `Duration` property of the Android `VideoView` reports a valid duration in milliseconds when the `Prepared` event of `VideoView` is fired. The Android `VideoPlayerRenderer` class uses that handler to obtain the `Duration` property:

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            ...
            ((IVideoPlayerController)Element).Duration = TimeSpan.FromMilliseconds(videoView.Duration);
        }
        ...
    }
}
```

### Video duration in UWP

The `NaturalDuration` property of `MediaElement` is a `TimeSpan` value and becomes valid when `MediaElement` fires the `MediaOpened` event:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        void OnMediaElementMediaOpened(object sender, RoutedEventArgs args)
        {
            ((IVideoPlayerController)Element).Duration = Control.NaturalDuration.TimeSpan;
        }
        ...
    }
}

```

## The Position property

`VideoPlayer` also needs a `Position` property that increases from zero to `Duration` as the video is playing. `VideoPlayer` implements this property like the `Position` property in the UWP `MediaElement`, which is a normal bindable property with public `set` and `get` accessors:

```

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Position property
        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(VideoPlayer), new TimeSpan(),
                propertyChanged: (bindable, oldValue, newValue) => ((VideoPlayer)bindable).SetTimeToEnd());

        public TimeSpan Position
        {
            set { SetValue(PositionProperty, value); }
            get { return (TimeSpan)GetValue(PositionProperty); }
        }
        ...
    }
}

```

The `get` accessor returns the current position of the video as it is playing, but the `set` accessor is intended to respond to the user's manipulation of the position bar by moving the video position forwards or backwards.

In iOS and Android, the property that obtains the current position has only a `get` accessor, and a `Seek` method is available to perform this second task. If you think about it, a separate `Seek` method seems to be a more sensible approach than a single `Position` property. A single `Position` property has an inherent problem: As the video plays, the `Position` property must be continually updated to reflect the new position. But you don't want most changes to the `Position` property to cause the video player to move to a new position in the video. If that happens, the video player would respond by seeking to the last value of the `Position` property, and the video wouldn't advance.

Despite the difficulties of implementing a `Position` property with `set` and `get` accessors, this approach was chosen because it is consistent with the UWP `MediaElement`, and it has a big advantage with data binding: The `Position` property of the `VideoPlayer` can be bound to the slider that is used both to display the position and to seek to a new position. However, several precautions are necessary when implementing this `Position` property to avoid feedback loops.

### Setting and getting iOS position

In iOS, the `CurrentTime` property of the `AVPlayerItem` object indicates the current position of the playing video.

The iOS `VideoPlayerRenderer` sets the `Position` property in the `UpdateStatus` handler at the same time that it sets the `Duration` property:

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            if (playerItem != null)
            {
                ...
                ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty,
ConvertTime(playerItem.CurrentTime));
            }
            ...
        }
    }
}
```

The renderer detects when the `Position` property set from `VideoPlayer` has changed in the `OnElementPropertyChanged` override, and uses that new value to call a `Seek` method on the `AVPlayer` object:

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                TimeSpan controlPosition = ConvertTime(player.CurrentTime);

                if (Math.Abs((controlPosition - Element.Position).TotalSeconds) > 1)
                {
                    player.Seek(CMTime.FromSeconds(Element.Position.TotalSeconds, 1));
                }
            }
            ...
        }
    }
}
```

Keep in mind that every time the `Position` property in `videoPlayer` is set from the `OnUpdateStatus` handler, the `Position` property fires a `PropertyChanged` event, which is detected in the `OnElementPropertyChanged` override. For most of these changes, the `OnElementPropertyChanged` method should do nothing. Otherwise, with every change in the video's position, it would be moved to the same position it just reached!

To avoid this feedback loop, the `OnElementPropertyChanged` method only calls `Seek` when the difference between the `Position` property and the current position of the `AVPlayer` is greater than one second.

## Setting and getting Android position

Just as in the iOS renderer, the Android `VideoPlayerRenderer` sets a new value for the `Position` property in the `OnUpdateStatus` handler. The `CurrentPosition` property of `VideoView` contains the new position in units of milliseconds:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            TimeSpan timeSpan = TimeSpan.FromMilliseconds(videoView.CurrentPosition);
            ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty, timeSpan);
        }
        ...
    }
}

```

Also, just as in the iOS renderer, the Android renderer calls the `SeekTo` method of `VideoView` when the `Position` property has changed, but only when the change is more than one second different from the `CurrentPosition` value of `VideoView`:

```

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                if (Math.Abs(videoView.CurrentPosition - Element.Position.TotalMilliseconds) > 1000)
                {
                    videoView.SeekTo((int)Element.Position.TotalMilliseconds);
                }
            }
            ...
        }
    }
}

```

## Setting and getting UWP position

The UWP `VideoPlayerRenderer` handles the `Position` in the same way as the iOS and Android renderers, but because the `Position` property of the UWP `MediaElement` is also a `TimeSpan` value, no conversion is necessary:

```

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                if (Math.Abs((Control.Position - Element.Position).TotalSeconds) > 1)
                {
                    Control.Position = Element.Position;
                }
            }
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty,
Control.Position);
        }
        ...
    }
}

```

## Calculating a TimeToEnd property

Sometimes video players show the time remaining in the video. This value begins at the video's duration when the video begins and decreases down to zero when the video ends.

`VideoPlayer` includes a read-only `TimeToEnd` property that is handled entirely within the class based on changes to the `Duration` and `Position` properties:

```

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        private static readonly BindablePropertyKey TimeToEndPropertyKey =
BindableProperty.CreateReadOnly(nameof(TimeToEnd), typeof(TimeSpan), typeof(VideoPlayer), new
TimeSpan());

        public static readonly BindableProperty TimeToEndProperty = TimeToEndPropertyKey.BindableProperty;

        public TimeSpan TimeToEnd
        {
            private set { SetValue(TimeToEndPropertyKey, value); }
            get { return (TimeSpan)GetValue(TimeToEndProperty); }
        }

        void SetTimeToEnd()
        {
            TimeToEnd = Duration - Position;
        }
        ...
    }
}

```

The `SetTimeToEnd` method is called from the property-changed handlers of both `Duration` and `Position`.

## A custom slider for video

It's possible to write a custom control for a position bar, or to use the `Xamarin.Forms Slider` or a class that derives from `Slider`, such as the following `PositionSlider` class. The class defines two new properties named `Duration` and `Position` of type `TimeSpan` that are intended to be data-bound to the two properties of the same name in `VideoPlayer`. Notice that the default binding mode of the `Position` property is two-way:

```
namespace FormsVideoLibrary
{
    public class PositionSlider : Slider
    {
        public static readonly BindableProperty DurationProperty =
            BindableProperty.Create(nameof(Duration), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(1),
            propertyChanged: (bindable, oldValue, newValue) =>
            {
                double seconds = ((TimeSpan)newValue).TotalSeconds;
                ((PositionSlider)bindable).Maximum = seconds <= 0 ? 1 : seconds;
            });

        public TimeSpan Duration
        {
            set { SetValue(DurationProperty, value); }
            get { return (TimeSpan)GetValue(DurationProperty); }
        }

        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(0),
            defaultBindingMode: BindingMode.TwoWay,
            propertyChanged: (bindable, oldValue, newValue) =>
            {
                double seconds = ((TimeSpan)newValue).TotalSeconds;
                ((PositionSlider)bindable).Value = seconds;
            });

        public TimeSpan Position
        {
            set { SetValue(PositionProperty, value); }
            get { return (TimeSpan)GetValue(PositionProperty); }
        }

        public PositionSlider()
        {
            PropertyChanged += (sender, args) =>
            {
                if (args.PropertyName == "Value")
                {
                    TimeSpan newPosition = TimeSpan.FromSeconds(Value);

                    if (Math.Abs(newPosition.TotalSeconds - Position.TotalSeconds) / Duration.TotalSeconds >
0.01)
                    {
                        Position = newPosition;
                    }
                }
            };
        }
    }
}
```

The property-changed handler for the `Duration` property sets the `Maximum` property of the underlying `Slider` to the `TotalSeconds` property of the `TimeSpan` value. Similarly, the property-changed handler for `Position` sets the `Value` property of the `Slider`. In this way, the underlying `Slider` tracks the position of the `PositionSlider`.

The `PositionSlider` is updated from the underlying `Slider` in only one instance: When the user manipulates the `Slider` to indicate that the video should be advanced or reversed to a new position. This is detected in the `PropertyChanged` handler in the constructor of the `PositionSlider`. The handler checks for a change in the `value` property, and if it's different from the `Position` property, then the `Position` property is set from the `Value` property.

In theory, the inner `if` statement could be written like this:

```
if (newPosition.Seconds != Position.Seconds)
{
    Position = newPosition;
}
```

However, the Android implementation of `Slider` has only 1,000 discrete steps regardless of the `Minimum` and `Maximum` settings. If the length of a video is greater than 1,000 seconds, then two different `Position` values would correspond to the same `Value` setting of the `slider`, and this `if` statement would trigger a false positive for a user manipulation of the `slider`. It's safer to instead check that the new position and existing position are greater than one-hundredth of the overall duration.

## Using the PositionSlider

Documentation for the UWP `MediaElement` warns about binding to the `Position` property because the property frequently updates. The documentation recommends that a timer be used to query the `Position` property.

That's a good recommendation, but the three `videoPlayerRenderer` classes are already indirectly using a timer to update the `Position` property. The `Position` property is changed in a handler for the `UpdateStatus` event, which is fired only 10 times a second.

Therefore, the `Position` property of the `VideoPlayer` can be bound to the `Position` property of the `PositionSlider` without performance issues, as demonstrated in the **Custom Position Bar** page:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    x:Class="VideoPlayerDemos.CustomPositionBarPage"
    Title="Custom Position Bar">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
            Grid.Row="0"
            AreTransportControlsEnabled="False"
            Source="{StaticResource ElephantsDream}" />

        ...
    
```

```

        <StackLayout Grid.Row="1"
            Orientation="Horizontal"
            Margin="10, 0"
            BindingContext="{x:Reference videoPlayer}">

            <Label Text="{Binding Path=Position,
                StringFormat='{0:hh\\:mm\\:ss}'}"
                VerticalOptions="Center"/>
    
```

```

        ...
    
```

```

        <Label Text="{Binding Path=TimeToEnd,
            StringFormat='{0:hh\\:mm\\:ss}'}"
                VerticalOptions="Center" />
    
```

```

</StackLayout>

```

```

<video:PositionSlider Grid.Row="2"
            Margin="10, 0, 10, 10"
            BindingContext="{x:Reference videoPlayer}"
            Duration="{Binding Duration}"
            Position="{Binding Position}">
    
```

```

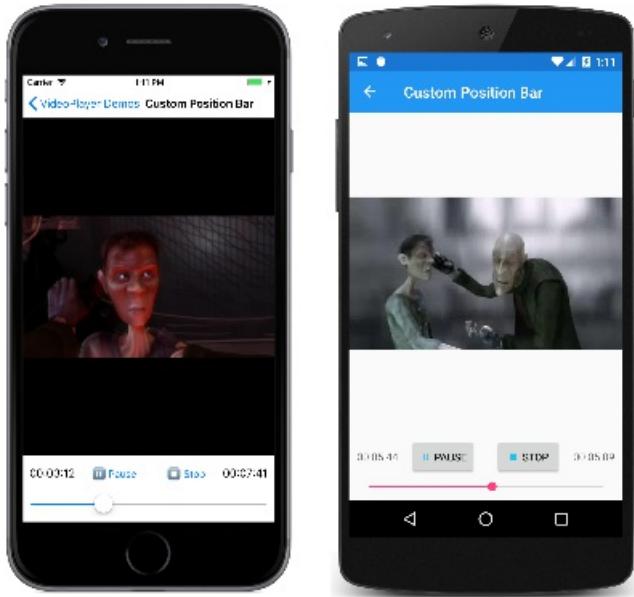
        <video:PositionSlider.Triggers>
            <DataTrigger TargetType="video:PositionSlider"
                Binding="{Binding Status}"
                Value="{x:Static video:VideoStatus.NotReady}">
                <Setter Property="IsEnabled" Value="False" />
            </DataTrigger>
        </video:PositionSlider.Triggers>
    
```

```

</video:PositionSlider>
</Grid>

```

The first ellipsis (...) hides the `ActivityIndicator`; it's the same as in the previous **Custom Transport** page. Notice the two `Label` elements displaying the `Position` and `TimeToEnd` properties. The ellipsis between those two `Label` elements hides the two `Button` elements shown in the **Custom Transport** page for Play, Pause, and Stop. The code-behind logic is also the same as the **Custom Transport** page.



This concludes the discussion of the `VideoPlayer`.

## Related Links

- [Video Player Demos \(sample\)](#)

# Xamarin.Forms Data Binding

10/23/2018 • 2 minutes to read • [Edit Online](#)

*Data binding is the technique of linking properties of two objects so that changes in one property are automatically reflected in the other property. Data binding is an integral part of the Model-View-ViewModel (MVVM) application architecture.*

## The Data Linking Problem

A Xamarin.Forms application consists of one or more pages, each of which generally contains multiple user-interface objects called *views*. One of the primary tasks of the program is to keep these views synchronized, and to keep track of the various values or selections that they represent. Often the views represent values from an underlying data source, and the user manipulates these views to change that data. When the view changes, the underlying data must reflect that change, and similarly, when the underlying data changes, that change must be reflected in the view.

To handle this job successfully, the program must be notified of changes in these views or the underlying data. The common solution is to define events that signal when a change occurs. An event handler can then be installed that is notified of these changes. It responds by transferring data from one object to another. However, when there are many views, there must also be many event handlers, and a lot of code gets involved.

## The Data Binding Solution

Data binding automates this job, and renders the event handlers unnecessary. (The events are still necessary, however, because the data-binding infrastructure uses them.) Data bindings can be implemented either in code or in XAML, but they are much more common in XAML where they help to reduce the size of the code-behind file. By replacing procedural code in event handlers with declarative code or markup, the application is simplified and clarified.

One of the two objects involved in a data binding is almost always an element that derives from `View` and forms part of the visual interface of a page. The other object is either:

- Another `View` derivative, usually on the same page.
- An object in a code file.

In demonstration programs such as those in the [DataBindingDemos](#) sample, data bindings between two `View` derivatives are often shown for purposes of clarity and simplicity. However, the same principles can be applied to data bindings between a `View` and other objects. When an application is built using the Model-View-ViewModel (MVVM) architecture, the class with underlying data is often called a *ViewModel*.

Data bindings are explored in the following series of articles:

## Basic Bindings

Learn the difference between the data binding target and source, and see simple data bindings in code and XAML.

## Binding Mode

Discover how the binding mode can control the flow of data between the two objects.

## String Formatting

Use a data binding to format and display objects as strings.

## Binding Path

Dive deeper into the `Path` property of the data binding to access sub-properties and collection members.

## Binding Value Converters

Use binding value converters to alter values within the data binding.

## Binding Fallbacks

Make data bindings more robust by defining fallback values to use if the binding process fails.

## The Command Interface

Implement the `Command` property with data bindings.

## Compiled Bindings

Use compiled bindings to improve data binding performance.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)
- [XAML Markup Extensions](#)

# Xamarin.Forms Basic Bindings

10/31/2018 • 9 minutes to read • [Edit Online](#)

A Xamarin.Forms data binding links a pair of properties between two objects, at least one of which is usually a user-interface object. These two objects are called the *target* and the *source*:

- The *target* is the object (and property) on which the data binding is set.
- The *source* is the object (and property) referenced by the data binding.

This distinction can sometimes be a little confusing: In the simplest case, data flows from the source to the target, which means that the value of the target property is set from the value of the source property. However, in some cases, data can alternatively flow from the target to the source, or in both directions. To avoid confusion, keep in mind that the target is always the object on which the data binding is set even if it's providing data rather than receiving data.

## Bindings with a Binding Context

Although data bindings are usually specified entirely in XAML, it's instructive to see data bindings in code. The **Basic Code Binding** page contains a XAML file with a `Label` and a `Slider`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.BasicCodeBindingPage"
    Title="Basic Code Binding">
    <StackLayout Padding="10, 0">
        <Label x:Name="label"
            Text="TEXT"
            FontSize="48"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The `Slider` is set for a range of 0 to 360. The intent of this program is to rotate the `Label` by manipulating the `Slider`.

Without data bindings, you would set the `ValueChanged` event of the `Slider` to an event handler that accesses the `Value` property of the `Slider` and sets that value to the `Rotation` property of the `Label`. The data binding automates that job; the event handler and the code within it are no longer necessary.

You can set a binding on an instance of any class that derives from `BindableObject`, which includes `Element`, `VisualElement`, `View`, and `View` derivatives. The binding is always set on the target object. The binding references the source object. To set the data binding, use the following two members of the target class:

- The `BindingContext` property specifies the source object.
- The `SetBinding` method specifies the target property and source property.

In this example, the `Label` is the binding target, and the `Slider` is the binding source. Changes in the `Slider` source affect the rotation of the `Label` target. Data flows from the source to the target.

The `SetBinding` method defined by `BindableObject` has an argument of type `BindingBase` from which the `Binding` class derives, but there are other `SetBinding` methods defined by the `BindableObjectExtensions` class. The code-behind file in the **Basic Code Binding** sample uses a simpler `SetBinding` extension method from this class.

```
public partial class BasicCodeBindingPage : ContentPage
{
    public BasicCodeBindingPage()
    {
        InitializeComponent();

        label.BindingContext = slider;
        label.SetBinding(Label.RotationProperty, "Value");
    }
}
```

The `Label` object is the binding target so that's the object on which this property is set and on which the method is called. The `BindingContext` property indicates the binding source, which is the `Slider`.

The `SetBinding` method is called on the binding target but specifies both the target property and the source property. The target property is specified as a `BindableProperty` object: `Label.RotationProperty`. The source property is specified as a string and indicates the `Value` property of `Slider`.

The `SetBinding` method reveals one of the most important rules of data bindings:

*The target property must be backed by a bindable property.*

This rule implies that the target object must be an instance of a class that derives from `BindableObject`. See the [Bindable Properties](#) article for an overview of bindable objects and bindable properties.

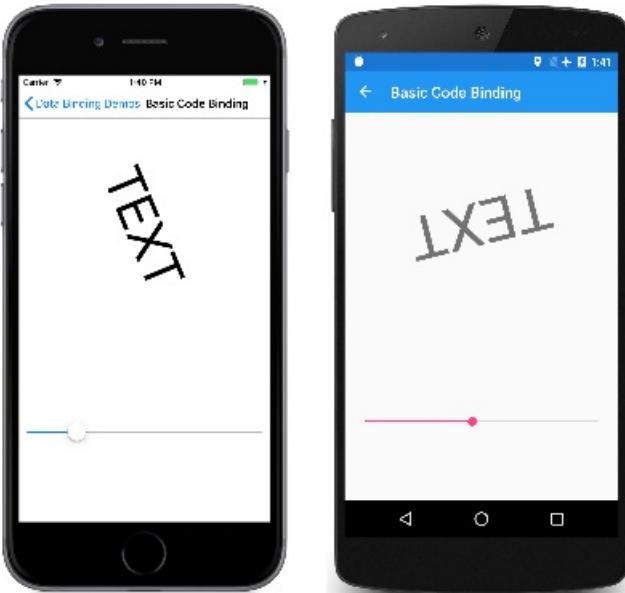
There is no such rule for the source property, which is specified as a string. Internally, reflection is used to access the actual property. In this particular case, however, the `value` property is also backed by a bindable property.

The code can be simplified somewhat: The `RotationProperty` bindable property is defined by `VisualElement`, and inherited by `Label` and `ContentPage` as well, so the class name isn't required in the `SetBinding` call:

```
label.SetBinding(RotationProperty, "Value");
```

However, including the class name is a good reminder of the target object.

As you manipulate the `Slider`, the `Label` rotates accordingly:



The **Basic Xaml Binding** page is identical to **Basic Code Binding** except that it defines the entire data binding in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.BasicXamlBindingPage"
    Title="Basic XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
            FontSize="80"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BindingContext="{x:Reference Name=slider}"
            Rotation="{Binding Path=Value}" />

        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

Just as in code, the data binding is set on the target object, which is the `Label`. Two XAML markup extensions are involved. These are instantly recognizable by the curly brace delimiters:

- The `x:Reference` markup extension is required to reference the source object, which is the `Slider` named `slider`.
- The `Binding` markup extension links the `Rotation` property of the `Label` to the `Value` property of the `Slider`.

See the article [XAML Markup Extensions](#) for more information about XAML markup extensions. The `x:Reference` markup extension is supported by the `ReferenceExtension` class; `Binding` is supported by the `BindingExtension` class. As the XML namespace prefixes indicate, `x:Reference` is part of the XAML 2009 specification, while `Binding` is part of Xamarin.Forms. Notice that no quotation marks appear within the curly braces.

It's easy to forget the `x:Reference` markup extension when setting the `BindingContext`. It's common to mistakenly set the property directly to the name of the binding source like this:

```
BindingContext="slider"
```

But that's not right. That markup sets the `BindingContext` property to a `string` object whose characters spell "slider"!

Notice that the source property is specified with the `Path` property of `BindingExtension`, which corresponds with the `Path` property of the `Binding` class.

The markup shown on the **Basic XAML Binding** page can be simplified: XAML markup extensions such as `x:Reference` and `Binding` can have *content property* attributes defined, which for XAML markup extensions means that the property name doesn't need to appear. The `Name` property is the content property of `x:Reference`, and the `Path` property is the content property of `Binding`, which means that they can be eliminated from the expressions:

```
<Label Text="TEXT"
    FontSize="80"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand"
    BindingContext="{x:Reference slider}"
    Rotation="{Binding Value}" />
```

## Bindings without a Binding Context

The `BindingContext` property is an important component of data bindings, but it is not always necessary. The source object can instead be specified in the `SetBinding` call or the `Binding` markup extension.

This is demonstrated in the **Alternative Code Binding** sample. The XAML file is similar to the **Basic Code Binding** sample except that the `Slider` is defined to control the `Scale` property of the `Label`. For that reason, the `Slider` is set for a range of -2 to 2:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.AlternativeCodeBindingPage"
    Title="Alternative Code Binding">
    <StackLayout Padding="10, 0">
        <Label x:Name="label"
            Text="TEXT"
            FontSize="40"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
            Minimum="-2"
            Maximum="2"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

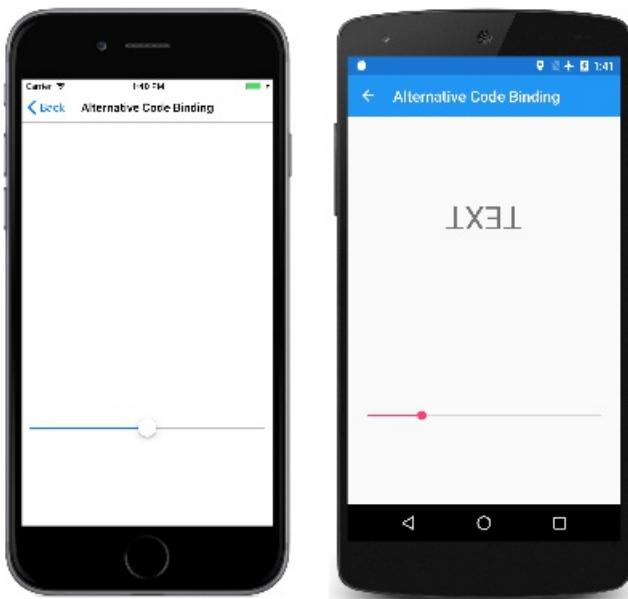
The code-behind file sets the binding with the `SetBinding` method defined by `BindableObject`. The argument is a `constructor` for the `Binding` class:

```
public partial class AlternativeCodeBindingPage : ContentPage
{
    public AlternativeCodeBindingPage()
    {
        InitializeComponent();

        label.SetBinding(Label.ScaleProperty, new Binding("Value", source: slider));
    }
}
```

The `Binding` constructor has 6 parameters, so the `source` parameter is specified with a named argument. The argument is the `slider` object.

Running this program might be a little surprising:



The iOS screen on the left shows how the screen looks when the page first appears. Where is the `Label`?

The problem is that the `slider` has an initial value of 0. This causes the `Scale` property of the `Label` to be also set to 0, overriding its default value of 1. This results in the `Label` being initially invisible. As the Android and Universal Windows Platform (UWP) screenshots demonstrate, you can manipulate the `Slider` to make the `Label` appear again, but its initial disappearance is disconcerting.

You'll discover in the [next article](#) how to avoid this problem by initializing the `Slider` from the default value of the `Scale` property.

#### NOTE

The `VisualElement` class also defines `ScaleX` and `ScaleY` properties, which can scale the `VisualElement` differently in the horizontal and vertical directions.

The **Alternative XAML Binding** page shows the equivalent binding entirely in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.AlternativeXamlBindingPage"
    Title="Alternative XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
            FontSize="40"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Scale="{Binding Source={x:Reference slider},
                Path=Value}" />

        <Slider x:Name="slider"
            Minimum="-2"
            Maximum="2"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

Now the `Binding` markup extension has two properties set, `Source` and `Path`, separated by a comma. They can appear on the same line if you prefer:

```
Scale="{Binding Source={x:Reference slider}, Path=Value}" />
```

The `Source` property is set to an embedded `x:Reference` markup extension that otherwise has the same syntax as setting the `BindingContext`. Notice that no quotation marks appear within the curly braces, and that the two properties must be separated by a comma.

The content property of the `Binding` markup extension is `Path`, but the `Path=` part of the markup extension can only be eliminated if it is the first property in the expression. To eliminate the `Path=` part, you need to swap the two properties:

```
Scale="{Binding Value, Source={x:Reference slider}}" />
```

Although XAML markup extensions are usually delimited by curly braces, they can also be expressed as object elements:

```
<Label Text="TEXT"
    FontSize="40"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Source="{x:Reference slider}"
            Path="Value" />
    </Label.Scale>
</Label>
```

Now the `Source` and `Path` properties are regular XAML attributes: The values appear within quotation marks and the attributes are not separated by a comma. The `x:Reference` markup extension can also become an object element:

```
<Label Text="TEXT"
    FontSize="40"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Path="Value">
            <Binding.Source>
                <x:Reference Name="slider" />
            </Binding.Source>
        </Binding>
    </Label.Scale>
</Label>
```

This syntax isn't common, but sometimes it's necessary when complex objects are involved.

The examples shown so far set the `BindingContext` property and the `Source` property of `Binding` to an `x:Reference` markup extension to reference another view on the page. These two properties are of type `Object`, and they can be set to any object that includes properties that are suitable for binding sources.

In the articles ahead, you'll discover that you can set the `BindingContext` or `Source` property to an `x:Static` markup extension to reference the value of a static property or field, or a `StaticResource` markup extension to reference an object stored in a resource dictionary, or directly to an object, which is generally (but not always) an instance of a `ViewModel`.

The `BindingContext` property can also be set to a `Binding` object so that the `Source` and `Path` properties of `Binding` define the binding context.

## Binding Context Inheritance

In this article, you've seen that you can specify the source object using the `BindingContext` property or the `Source` property of the `Binding` object. If both are set, the `Source` property of the `Binding` takes precedence over the `BindingContext`.

The `BindingContext` property has an extremely important characteristic:

*The setting of the `BindingContext` property is inherited through the visual tree.*

As you'll see, this can be very handy for simplifying binding expressions, and in some cases — particularly in Model-View-ViewModel (MVVM) scenarios — it is essential.

The **Binding Context Inheritance** sample is a simple demonstration of the inheritance of the binding context:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.BindingContextInheritancePage"
    Title="BindingContext Inheritance">
    <StackLayout Padding="10">

        <StackLayout VerticalOptions="FillAndExpand"
            BindingContext="{x:Reference slider}">

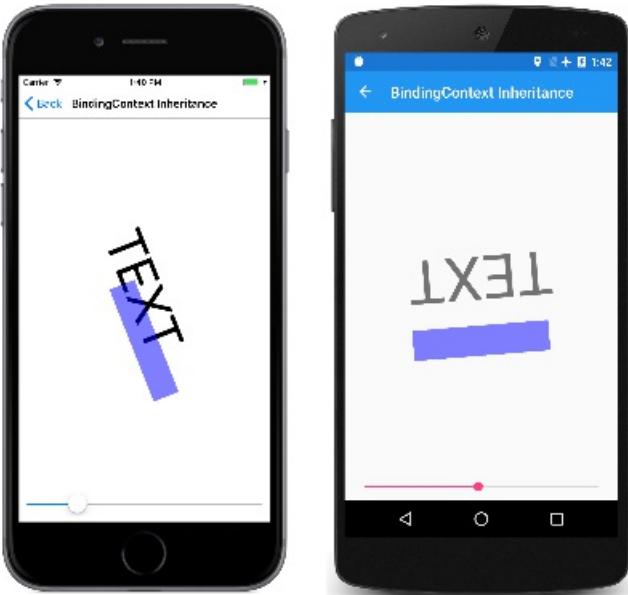
            <Label Text="TEXT"
                FontSize="80"
                HorizontalOptions="Center"
                VerticalOptions="EndAndExpand"
                Rotation="{Binding Value}" />

            <BoxView Color="#800000FF"
                WidthRequest="180"
                HeightRequest="40"
                HorizontalOptions="Center"
                VerticalOptions="StartAndExpand"
                Rotation="{Binding Value}" />
        </StackLayout>

        <Slider x:Name="slider"
            Maximum="360" />

    </StackLayout>
</ContentPage>
```

The `BindingContext` property of the `StackLayout` is set to the `slider` object. This binding context is inherited by both the `Label` and the `BoxView`, both of which have their `Rotation` properties set to the `Value` property of the `Slider`:



In the [next article](#), you'll see how the *binding mode* can change the flow of data between target and source objects.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms Binding Mode

11/12/2018 • 16 minutes to read • [Edit Online](#)

In the [previous article](#), the **Alternative Code Binding** and **Alternative XAML Binding** pages featured a `Label` with its `Scale` property bound to the `Value` property of a `Slider`. Because the `Slider` initial value is 0, this caused the `Scale` property of the `Label` to be set to 0 rather than 1, and the `Label` disappeared.

In the **DataBindingDemos** sample, the **Reverse Binding** page is similar to the programs in the previous article, except that the data binding is defined on the `Slider` rather than on the `Label`:

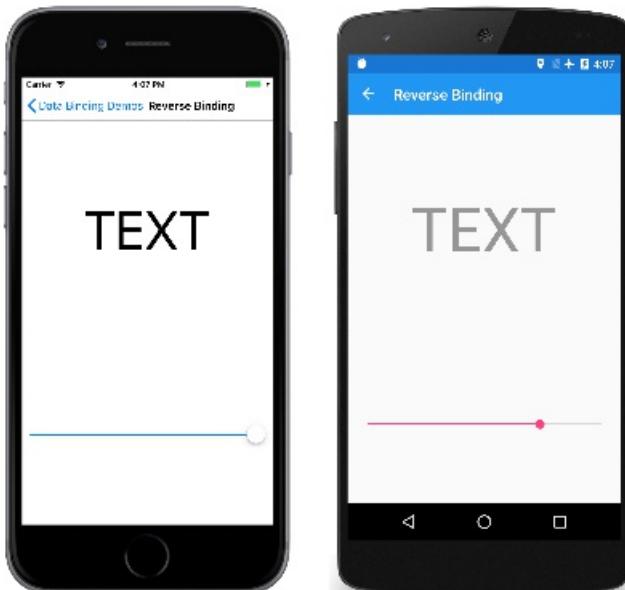
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.ReverseBindingPage"
    Title="Reverse Binding">
<StackLayout Padding="10, 0">

    <Label x:Name="label"
        Text="TEXT"
        FontSize="80"
        HorizontalOptions="Center"
        VerticalOptions="CenterAndExpand" />

    <Slider x:Name="slider"
        VerticalOptions="CenterAndExpand"
        Value="{Binding Source={x:Reference label},
            Path=Opacity}" />
</StackLayout>
</ContentPage>
```

At first, this might seem backwards: Now the `Label` is the data-binding source, and the `Slider` is the target. The binding references the `Opacity` property of the `Label`, which has a default value of 1.

As you might expect, the `slider` is initialized to the value 1 from the initial `opacity` value of `Label`. This is shown in the iOS screenshot on the left:



But you might be surprised that the `Slider` continues to work, as the Android and UWP screenshots demonstrate. This seems to suggest that the data binding works better when the `Slider` is the binding target

rather than the `Label` because the initialization works like we might expect.

The difference between the **Reverse Binding** sample and the earlier samples involves the *binding mode*.

## The Default Binding Mode

The binding mode is specified with a member of the `BindingMode` enumeration:

- `Default`
- `TwoWay` – data goes both ways between source and target
- `OneWay` – data goes from source to target
- `OneWayToSource` – data goes from target to source
- `OneTime` – data goes from source to target, but only when the `BindingContext` changes (new with Xamarin.Forms 3.0)

Every bindable property has a default binding mode that is set when the bindable property is created, and which is available from the `DefaultBindingMode` property of the `BindableProperty` object. This default binding mode indicates the mode in effect when that property is a data-binding target.

The default binding mode for most properties such as `Rotation`, `Scale`, and `Opacity` is `OneWay`. When these properties are data-binding targets, then the target property is set from the source.

However, the default binding mode for the `Value` property of `Slider` is `TwoWay`. This means that when the `Value` property is a data-binding target, then the target is set from the source (as usual) but the source is also set from the target. This is what allows the `Slider` to be set from the initial `Opacity` value.

This two-way binding might seem to create an infinite loop, but that doesn't happen. Bindable properties do not signal a property change unless the property actually changes. This prevents an infinite loop.

### Two-Way Bindings

Most bindable properties have a default binding mode of `OneWay` but the following properties have a default binding mode of `TwoWay`:

- `Date` property of `DatePicker`
- `Text` property of `Editor`, `Entry`, `SearchBar`, and `EntryCell`
- `IsRefreshing` property of `ListView`
- `SelectedItem` property of `MultiPage`
- `SelectedIndex` and `SelectedItem` properties of `Picker`
- `Value` property of `Slider` and `Stepper`
- `IsToggled` property of `Switch`
- `On` property of `SwitchCell`
- `Time` property of `TimePicker`

These particular properties are defined as `TwoWay` for a very good reason:

When data bindings are used with the Model-View-ViewModel (MVVM) application architecture, the `ViewModel` class is the data-binding source, and the `View`, which consists of views such as `Slider`, are data-binding targets. MVVM bindings resemble the **Reverse Binding** sample more than the bindings in the previous samples. It is very likely that you want each view on the page to be initialized with the value of the corresponding property in the `ViewModel`, but changes in the view should also affect the `ViewModel` property.

The properties with default binding modes of `TwoWay` are those properties most likely to be used in MVVM scenarios.

### One-Way-to-Source Bindings

Read-only bindable properties have a default binding mode of `OneWayToSource`. There is only one read/write bindable property that has a default binding mode of `OneWayToSource`:

- `SelectedItem` property of `ListView`

The rationale is that a binding on the `SelectedItem` property should result in setting the binding source. An example later in this article overrides that behavior.

## One-Time Bindings

Several properties have a default binding mode of `OneTime`. These are:

- `IsTextPredictionEnabled` property of `Entry`
- `Text`, `BackgroundColor`, and `Style` properties of `Span`.

Target properties with a binding mode of `OneTime` are updated only when the binding context changes. For bindings on these target properties, this simplifies the binding infrastructure because it is not necessary to monitor changes in the source properties.

## ViewModels and Property-Change Notifications

The [Simple Color Selector](#) page demonstrates the use of a simple ViewModel. Data bindings allow the user to select a color using three `Slider` elements for the hue, saturation, and luminosity.

The ViewModel is the data-binding source. The ViewModel does *not* define bindable properties, but it does implement a notification mechanism that allows the binding infrastructure to be notified when the value of a property changes. This notification mechanism is the `INotifyPropertyChanged` interface, which defines a single property named `PropertyChanged`. A class that implements this interface generally fires the event when one of its public properties changes value. The event does not need to be fired if the property never changes. (The `INotifyPropertyChanged` interface is also implemented by `BindableObject` and a `PropertyChanged` event is fired whenever a bindable property changes value.)

The `HslColorViewModel` class defines five properties: The `Hue`, `Saturation`, `Luminosity`, and `Color` properties are interrelated. When any one of the three color components changes value, the `Color` property is recalculated, and `PropertyChanged` events are fired for all four properties:

```
public class HslColorViewModel : INotifyPropertyChanged
{
    Color color;
    string name;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Hue
    {
        set
        {
            if (color.Hue != value)
            {
                Color = Color.FromHsla(value, color.Saturation, color.Luminosity);
            }
        }
        get
        {
            return color.Hue;
        }
    }

    public double Saturation
    {
        set
        {
            if (color.Saturation != value)
            {
                Color = Color.FromHsla(color.Hue, value, color.Luminosity);
            }
        }
        get
        {
            return color.Saturation;
        }
    }

    public double Luminosity
    {
        set
        {
            if (color.Luminosity != value)
            {
                Color = Color.FromHsla(color.Hue, color.Saturation, value);
            }
        }
        get
        {
            return color.Luminosity;
        }
    }

    public Color Color
    {
        get { return color; }
        set
        {
            if (color != value)
            {
                color = value;
                OnPropertyChanged("Color");
                OnPropertyChanged("Hue");
                OnPropertyChanged("Saturation");
                OnPropertyChanged("Luminosity");
            }
        }
    }
}
```

```

    {
        if (color.Saturation != value)
        {
            Color = Color.FromHsla(color.Hue, value, color.Luminosity);
        }
    }
    get
    {
        return color.Saturation;
    }
}

public double Luminosity
{
    set
    {
        if (color.Luminosity != value)
        {
            Color = Color.FromHsla(color.Hue, color.Saturation, value);
        }
    }
    get
    {
        return color.Luminosity;
    }
}

public Color Color
{
    set
    {
        if (color != value)
        {
            color = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Hue"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Saturation"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Luminosity"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));

            Name = NamedColor.GetNearestColorName(color);
        }
    }
    get
    {
        return color;
    }
}

public string Name
{
    private set
    {
        if (name != value)
        {
            name = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
        }
    }
    get
    {
        return name;
    }
}
}

```

When the `Color` property changes, the static `GetNearestColorName` method in the `NamedColor` class (also included in the **DataBindingDemos** solution) obtains the closest named color and sets the `Name` property. This `Name`

property has a private `set` accessor, so it cannot be set from outside the class.

When a ViewModel is set as a binding source, the binding infrastructure attaches a handler to the `PropertyChanged` event. In this way, the binding can be notified of changes to the properties, and can then set the target properties from the changed values.

However, when a target property (or the `Binding` definition on a target property) has a `BindingMode` of `OneTime`, it is not necessary for the binding infrastructure to attach a handler on the `PropertyChanged` event. The target property is updated only when the `BindingContext` changes and not when the source property itself changes.

The **Simple Color Selector** XAML file instantiates the `HslColorViewModel` in the page's resource dictionary and initializes the `color` property. The `BindingContext` property of the `Grid` is set to a `StaticResource` binding extension to reference that resource:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.SimpleColorSelectorPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <local:HslColorViewModel x:Key="viewModel"
                Color="MediumTurquoise" />

            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid BindingContext="{StaticResource viewModel}">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <BoxView Color="{Binding Color}"
            Grid.Row="0" />

        <StackLayout Grid.Row="1"
            Margin="10, 0">

            <Label Text="{Binding Name}"
                HorizontalTextAlignment="Center" />

            <Slider Value="{Binding Hue}" />

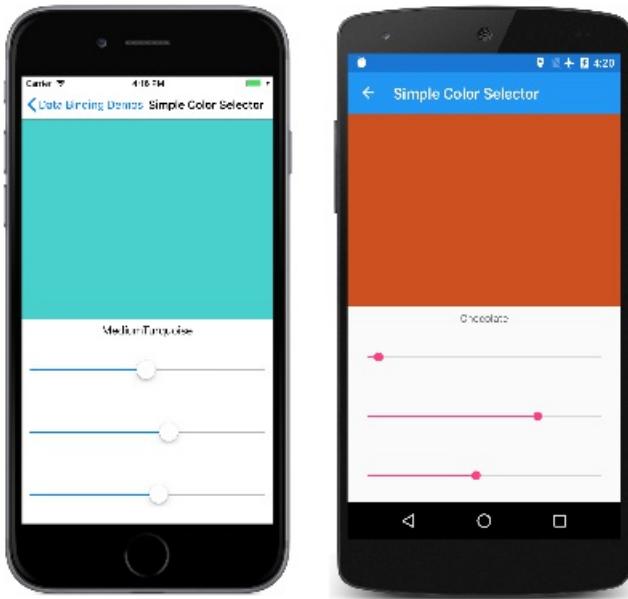
            <Slider Value="{Binding Saturation}" />

            <Slider Value="{Binding Luminosity}" />
        </StackLayout>
    </Grid>
</ContentPage>
```

The `BoxView`, `Label`, and three `Slider` views inherit the binding context from the `Grid`. These views are all binding targets that reference source properties in the ViewModel. For the `color` property of the `BoxView`, and the `Text` property of the `Label`, the data bindings are `oneWay`: The properties in the view are set from the properties in the ViewModel.

The `value` property of the `Slider`, however, is `TwoWay`. This allows each `Slider` to be set from the ViewModel, and also for the ViewModel to be set from each `Slider`.

When the program is first run, the `BoxView`, `Label`, and three `Slider` elements are all set from the ViewModel based on the initial `Color` property set when the ViewModel was instantiated. This is shown in the iOS screenshot at the left:



As you manipulate the sliders, the `BoxView` and `Label` are updated accordingly, as illustrated by the Android and UWP screenshots.

Instantiating the ViewModel in the resource dictionary is one common approach. It's also possible to instantiate the ViewModel within property element tags for the `BindingContext` property. In the **Simple Color Selector** XAML file, try removing the `HslColorViewModel` from the resource dictionary and set it to the `BindingContext` property of the `Grid` like this:

```
<Grid>
    <Grid.BindingContext>
        <local:HslColorViewModel Color="MediumTurquoise" />
    </Grid.BindingContext>

    ...
</Grid>
```

The binding context can be set in a variety of ways. Sometimes, the code-behind file instantiates the ViewModel and sets it to the `BindingContext` property of the page. These are all valid approaches.

## Overriding the Binding Mode

If the default binding mode on the target property is not suitable for a particular data binding, it's possible to override it by setting the `Mode` property of `Binding` (or the `Mode` property of the `Binding` markup extension) to one of the members of the `BindingMode` enumeration.

However, setting the `Mode` property to `TwoWay` doesn't always work as you might expect. For example, try modifying the **Alternative XAML Binding** XAML file to include `TwoWay` in the binding definition:

```
<Label Text="TEXT"
    FontSize="40"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand"
    Scale="{Binding Source={x:Reference slider},
        Path=Value,
        Mode=TwoWay}" />
```

It might be expected that the `Slider` would be initialized to the initial value of the `Scale` property, which is 1, but that doesn't happen. When a `TwoWay` binding is initialized, the target is set from the source first, which means that the `Scale` property is set to the `Slider` default value of 0. When the `TwoWay` binding is set on the `Slider`, then the `Slider` is initially set from the source.

You can set the binding mode to `OneWayToSource` in the **Alternative XAML Binding** sample:

```
<Label Text="TEXT"
    FontSize="40"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand"
    Scale="{Binding Source={x:Reference slider},
        Path=Value,
        Mode=OneWayToSource}" />
```

Now the `Slider` is initialized to 1 (the default value of `Scale`) but manipulating the `Slider` doesn't affect the `Scale` property, so this is not very useful.

#### NOTE

The `VisualElement` class also defines `ScaleX` and `ScaleY` properties, which can scale the `visualElement` differently in the horizontal and vertical directions.

A very useful application of overriding the default binding mode with `TwoWay` involves the `SelectedItem` property of `ListView`. The default binding mode is `OneWayToSource`. When a data binding is set on the `SelectedItem` property to reference a source property in a ViewModel, then that source property is set from the `ListView` selection. However, in some circumstances, you might also want the `ListView` to be initialized from the ViewModel.

The **Sample Settings** page demonstrates this technique. This page represents a simple implementation of application settings, which are very often defined in a ViewModel, such as this `SampleSettingsViewModel` file:

```
public class SampleSettingsViewModel : INotifyPropertyChanged
{
    string name;
    DateTime birthDate;
    bool codesInCSharp;
    double numberOfCopies;
    NamedColor backgroundNamedColor;

    public event PropertyChangedEventHandler PropertyChanged;

    public SampleSettingsViewModel(IDictionary<string, object> dictionary)
    {
        Name = GetDictionaryEntry<string>(dictionary, "Name");
        BirthDate = GetDictionaryEntry(dictionary, "BirthDate", new DateTime(1980, 1, 1));
        CodesInCSharp = GetDictionaryEntry<bool>(dictionary, "CodesInCSharp");
        NumberOfCopies = GetDictionaryEntry(dictionary, "NumberOfCopies", 1.0);
        BackgroundNamedColor = NamedColor.Find(GetDictionaryEntry(dictionary, "BackgroundNamedColor",
            "White"));
    }
}
```

```

}

public string Name
{
    set { SetProperty(ref name, value); }
    get { return name; }
}

public DateTime BirthDate
{
    set { SetProperty(ref birthDate, value); }
    get { return birthDate; }
}

public bool CodesInCSharp
{
    set { SetProperty(ref codesInCSharp, value); }
    get { return codesInCSharp; }
}

public double NumberOfCopies
{
    set { SetProperty(ref numberOfCopies, value); }
    get { return numberOfCopies; }
}

public NamedColor BackgroundNamedColor
{
    set
    {
        if (SetProperty(ref backgroundNamedColor, value))
        {
            OnPropertyChanged("BackgroundColor");
        }
    }
    get { return backgroundNamedColor; }
}

public Color BackgroundColor
{
    get { return BackgroundNamedColor?.Color ?? Color.White; }
}

public void SaveState(IDictionary<string, object> dictionary)
{
    dictionary["Name"] = Name;
    dictionary["BirthDate"] = BirthDate;
    dictionary["CodesInCSharp"] = CodesInCSharp;
    dictionary["NumberOfCopies"] = NumberOfCopies;
    dictionary["BackgroundNamedColor"] = BackgroundNamedColor.Name;
}

T GetDictionaryEntry<T>(IDictionary<string, object> dictionary, string key, T defaultValue = default(T))
{
    return dictionary.ContainsKey(key) ? (T)dictionary[key] : defaultValue;
}

bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
{
    if (Object.Equals(storage, value))
        return false;

    storage = value;
    OnPropertyChanged(propertyName);
    return true;
}

protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
}

```

```
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Each application setting is a property that is saved to the `Xamarin.Forms` properties dictionary in a method named `SaveState` and loaded from that dictionary in the constructor. Towards the bottom of the class are two methods that help streamline ViewModels and make them less prone to errors. The `OnPropertyChanged` method at the bottom has an optional parameter that is set to the calling property. This avoids spelling errors when specifying the name of the property as a string.

The  `SetProperty` method in the class does even more: It compares the value that is being set to the property with the value stored as a field, and only calls `OnPropertyChanged` when the two values are not equal.

The `SampleSettingsViewModel` class defines two properties for the background color: The `BackgroundNamedColor` property is of type `NamedColor`, which is a class also included in the **DataBindingDemos** solution. The `BackgroundColor` property is of type `Color`, and is obtained from the `Color` property of the `NamedColor` object.

The `NamedColor` class uses .NET reflection to enumerate all the static public fields in the `Xamarin.Forms` `Color` structure, and to store them with their names in a collection accessible from the static `All` property:

```
public class NamedColor : IEquatable<NamedColor>, IComparable<NamedColor>
{
    // Instance members
    private NamedColor()
    {
    }

    public string Name { private set; get; }

    public string FriendlyName { private set; get; }

    public Color Color { private set; get; }

    public string RgbDisplay { private set; get; }

    public bool Equals(NamedColor other)
    {
        return Name.Equals(other.Name);
    }

    public int CompareTo(NamedColor other)
    {
        return Name.CompareTo(other.Name);
    }

    // Static members
    static NamedColor()
    {
        List<NamedColor> all = new List<NamedColor>();
        StringBuilder stringBuilder = new StringBuilder();

        // Loop through the public static fields of the Color structure.
        foreach (FieldInfo fieldInfo in typeof(Color).GetRuntimeFields())
        {
            if (fieldInfo.IsPublic &&
                fieldInfo.IsStatic &&
                fieldInfo.FieldType == typeof(Color))
            {
                // Convert the name to a friendly name.
                string name = fieldInfo.Name;
                stringBuilder.Clear();
                int index = 0;

                foreach (char ch in name)

```

```

        {
            if (index != 0 && Char.IsUpper(ch))
            {
                stringBuilder.Append(' ');
            }
            stringBuilder.Append(ch);
            index++;
        }

        // Instantiate a NamedColor object.
        Color color = (Color)fieldInfo.GetValue(null);

        NamedColor namedColor = new NamedColor
        {
            Name = name,
            FriendlyName = stringBuilder.ToString(),
            Color = color,
            RgbDisplay = String.Format("{0:X2}-{1:X2}-{2:X2}",
                (int)(255 * color.R),
                (int)(255 * color.G),
                (int)(255 * color.B))
        };

        // Add it to the collection.
        all.Add(namedColor);
    }
}

all.TrimExcess();
all.Sort();
All = all;
}

public static IList<NamedColor> All { private set; get; }

public static NamedColor Find(string name)
{
    return ((List<NamedColor>)All).Find(nc => nc.Name == name);
}

public static string GetNearestColorName(Color color)
{
    double shortestDistance = 1000;
    NamedColor closestColor = null;

    foreach (NamedColor namedColor in NamedColor.All)
    {
        double distance = Math.Sqrt(Math.Pow(color.R - namedColor.Color.R, 2) +
            Math.Pow(color.G - namedColor.Color.G, 2) +
            Math.Pow(color.B - namedColor.Color.B, 2));

        if (distance < shortestDistance)
        {
            shortestDistance = distance;
            closestColor = namedColor;
        }
    }
    return closestColor.Name;
}
}

```

The `App` class in the **DataBindingDemos** project defines a property named `Settings` of type `SampleSettingsViewModel`. This property is initialized when the `App` class is instantiated, and the `SaveState` method is called when the `OnSleep` method is called:

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();

        Settings = new SampleSettingsViewModel(�.Current.Properties);

        MainPage = new NavigationPage(new MainPage());
    }

    public SampleSettingsViewModel Settings { private set; get; }

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

    protected override void OnSleep()
    {
        // Handle when your app sleeps
        Settings.SaveState(�.Current.Properties);
    }

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

For more information on the application lifecycle methods, see the article [App Lifecycle](#).

Almost everything else is handled in the `SampleSettingsPage.xaml` file. The `BindingContext` of the page is set using a `Binding` markup extension: The binding source is the static `Application.Current` property, which is the instance of the `App` class in the project, and the `Path` is set to the `Settings` property, which is the `SampleSettingsViewModel` object:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.SampleSettingsPage"
    Title="Sample Settings"
    BindingContext="{Binding Source={x:Static Application.Current},
    Path=Settings}">

<StackLayout BackgroundColor="{Binding BackgroundColor}"
    Padding="10"
    Spacing="10">

    <StackLayout Orientation="Horizontal">
        <Label Text="Name: "
            VerticalOptions="Center" />

        <Entry Text="{Binding Name}"
            Placeholder="your name"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="Center" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <Label Text="Birth Date: "
            VerticalOptions="Center" />

        <DatePicker Date="{Binding BirthDate}"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>
```

```

        VerticalOptions="Center" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <Label Text="Do you code in C#? "
            VerticalOptions="Center" />

        <Switch IsToggled="{Binding CodesInCSharp}"
            VerticalOptions="Center" />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
        <Label Text="Number of Copies: "
            VerticalOptions="Center" />

        <Stepper Value="{Binding NumberOfCopies}"
            VerticalOptions="Center" />
    </StackLayout>

    <Label Text="{Binding NumberOfCopies}"
        VerticalOptions="Center" />
</StackLayout>

<Label Text="Background Color:" />

<ListView x:Name="colorListView"
    ItemsSource="{x:Static local:NamedColor.All}"
    SelectedItem="{Binding BackgroundNamedColor, Mode=TwoWay}"
    VerticalOptions="FillAndExpand"
    RowHeight="40">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <StackLayout Orientation="Horizontal">
                    <BoxView Color="{Binding Color}"
                        HeightRequest="32"
                        WidthRequest="32"
                        VerticalOptions="Center" />

                    <Label Text="{Binding FriendlyName}"
                        FontSize="24"
                        VerticalOptions="Center" />
                </StackLayout>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
</StackLayout>
</ContentPage>
```

All the children of the page inherit the binding context. Most of the other bindings on this page are to properties in `SampleSettingsViewModel`. The `BackgroundColor` property is used to set the `BackgroundColor` property of the `StackLayout`, and the `Entry`, `DatePicker`, `Switch`, and `Stepper` properties are all bound to other properties in the `ViewModel`.

The `ItemsSource` property of the `ListView` is set to the static `NamedColor.All` property. This fills the `ListView` with all the `NamedColor` instances. For each item in the `ListView`, the binding context for the item is set to a `NamedColor` object. The `BoxView` and `Label` in the `ViewCell` are bound to properties in `NamedColor`.

The `SelectedItem` property of the `ListView` is of type `NamedColor`, and is bound to the `BackgroundNamedColor` property of `SampleSettingsViewModel`:

```
SelectedItem="{Binding BackgroundNamedColor, Mode=TwoWay}"
```

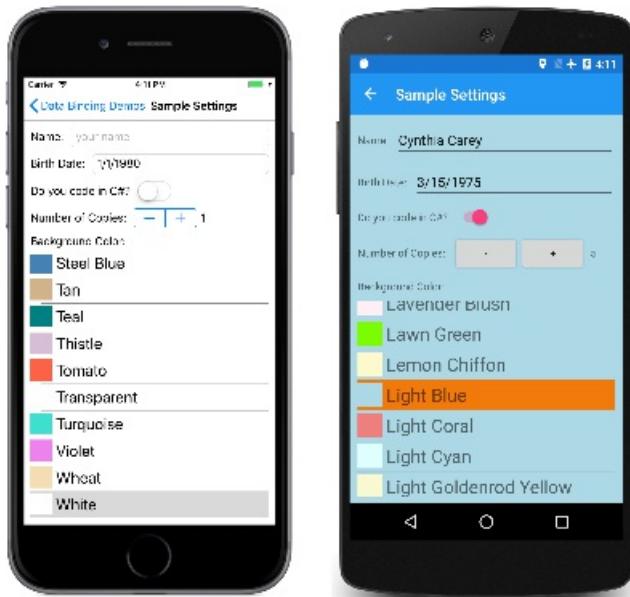
The default binding mode for `SelectedItem` is `OneWayToSource`, which sets the `ViewModel` property from the selected item. The `TwoWay` mode allows the `SelectedItem` to be initialized from the `ViewModel`.

However, when the `SelectedItem` is set in this way, the `ListView` does not automatically scroll to show the selected item. A little code in the code-behind file is necessary:

```
public partial class SampleSettingsPage : ContentPage
{
    public SampleSettingsPage()
    {
        InitializeComponent();

        if (colorListView.SelectedItem != null)
        {
            colorListView.ScrollTo(colorListView.SelectedItem,
                ScrollToPosition.MakeVisible,
                false);
        }
    }
}
```

The iOS screenshot at the left shows the program when it's first run. The constructor in `SampleSettingsViewModel` initializes the background color to white, and that's what's selected in the `ListView`:



The other two screenshots show altered settings. When experimenting with this page, remember to put the program to sleep or to terminate it on the device or emulator that it's running. Terminating the program from the Visual Studio debugger will not cause the `OnSleep` override in the `App` class to be called.

In the next article you'll see how to specify **String Formatting** of data bindings that are set on the `Text` property of `Label`.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms String Formatting

11/20/2018 • 4 minutes to read • [Edit Online](#)

Sometimes it's convenient to use data bindings to display the string representation of an object or value. For example, you might want to use a `Label` to display the current value of a `Slider`. In this data binding, the `Slider` is the source, and the target is the `Text` property of the `Label`.

When displaying strings in code, the most powerful tool is the static `String.Format` method. The formatting string includes formatting codes specific to various types of objects, and you can include other text along with the values being formatted. See the [Formatting Types in .NET](#) article for more information on string formatting.

## The `StringFormat` Property

This facility is carried over into data bindings: You set the `StringFormat` property of `Binding` (or the `StringFormat` property of the `Binding` markup extension) to a standard .NET formatting string with one placeholder:

```
<Slider x:Name="slider" />
<Label Text="{Binding Source={x:Reference slider},
                     Path=Value,
                     StringFormat='The slider value is {0:F2}'}" />
```

Notice that the formatting string is delimited by single-quote (apostrophe) characters to help the XAML parser avoid treating the curly braces as another XAML markup extension. Otherwise, that string without the single-quote character is the same string you'd use to display a floating-point value in a call to `String.Format`. A formatting specification of `F2` causes the value to be displayed with two decimal places.

The `StringFormat` property only makes sense when the target property is of type `string`, and the binding mode is `OneWay` or `TwoWay`. For two-way bindings, the `StringFormat` is only applicable for values passing from the source to the target.

As you'll see in the next article on the [Binding Path](#), data bindings can become quite complex and convoluted. When debugging these data bindings, you can add a `Label` into the XAML file with a `StringFormat` to display some intermediate results. Even if you use it only to display an object's type, that can be helpful.

The [String Formatting](#) page illustrates several uses of the `StringFormat` property:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="DataBindingDemos.StringFormattingPage"
    Title="String Formatting">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <Style TargetType="BoxView">
                <Setter Property="Color" Value="Blue" />
                <Setter Property="HeightRequest" Value="2" />
                <Setter Property="Margin" Value="0, 5" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Margin="10">
        <Slider x:Name="slider" />
        <Label Text="{Binding Source={x:Reference slider},
            Path=Value,
            StringFormat='The slider value is {0:F2}'}" />

        <BoxView />

        <TimePicker x:Name="timePicker" />
        <Label Text="{Binding Source={x:Reference timePicker},
            Path=Time,
            StringFormat='The TimeSpan is {0:c}'}" />

        <BoxView />

        <Entry x:Name="entry" />
        <Label Text="{Binding Source={x:Reference entry},
            Path=Text,
            StringFormat='The Entry text is \"{0}\"' }" />

        <BoxView />

        <StackLayout BindingContext="{x:Static sys:DateTime.Now}">
            <Label Text="{Binding}" />
            <Label Text="{Binding Path=Ticks,
                StringFormat='{0:N0} ticks since 1/1/1'}" />
            <Label Text="{Binding StringFormat='The {{0:MMMM}} specifier produces {0:MMMM}' }" />
            <Label Text="{Binding StringFormat='The long date is {0:D}' }" />
        </StackLayout>

        <BoxView />

        <StackLayout BindingContext="{x:Static sys:Math.PI}">
            <Label Text="{Binding}" />
            <Label Text="{Binding StringFormat='PI to 4 decimal points = {0:F4}' }" />
            <Label Text="{Binding StringFormat='PI in scientific notation = {0:E7}' }" />
        </StackLayout>
    </StackLayout>
</ContentPage>

```

The bindings on the `Slider` and `TimePicker` show the use of format specifications particular to `double` and `TimeSpan` data types. The `StringFormat` that displays the text from the `Entry` view demonstrates how to specify double quotation marks in the formatting string with the use of the `&quot;` HTML entity.

The next section in the XAML file is a `StackLayout` with a `BindingContext` set to an `x:Static` markup extension

that references the static `DateTime.Now` property. The first binding has no properties:

```
<Label Text="{Binding}" />
```

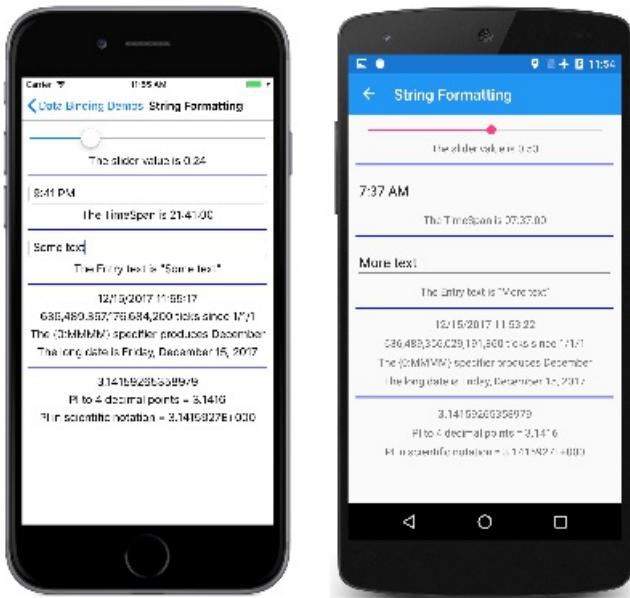
This simply displays the `DateTime` value of the `BindingContext` with default formatting. The second binding displays the `Ticks` property of `DateTime`, while the other two bindings display the `DateTime` itself with specific formatting. Notice this `StringFormat`:

```
<Label Text="{Binding StringFormat='The {{0:MMMM}} specifier produces {0:MMMM}'}" />
```

If you need to display left or right curly braces in your formatting string, simply use a pair of them.

The last section sets the `BindingContext` to the value of `Math.PI` and displays it with default formatting and two different types of numeric formatting.

Here's the program running:



## ViewModels and String Formatting

When you're using `Label` and `StringFormat` to display the value of a view that is also the target of a ViewModel, you can either define the binding from the view to the `Label` or from the ViewModel to the `Label`. In general, the second approach is best because it verifies that the bindings between the View and ViewModel are working.

This approach is shown in the **Better Color Selector** sample, which uses the same ViewModel as the **Simple Color Selector** program shown in the **Binding Mode** article:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.BetterColorSelectorPage"
    Title="Better Color Selector">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <StackLayout.BindingContext>
            <local:HslColorViewModel Color="Sienna" />
        </StackLayout.BindingContext>

        <BoxView Color="{Binding Color}"
            VerticalOptions="FillAndExpand" />

        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />

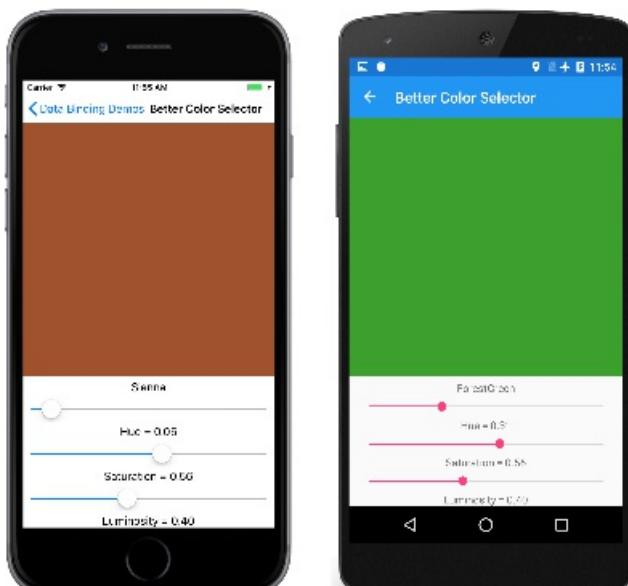
            <Slider Value="{Binding Hue}" />
            <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />

            <Slider Value="{Binding Saturation}" />
            <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />

            <Slider Value="{Binding Luminosity}" />
            <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>

```

There are now three pairs of `Slider` and `Label` elements that are bound to the same source property in the `HslColorViewModel` object. The only difference is that `Label` has a `StringFormat` property to display each `slider` value.



You might be wondering how you could display RGB (red, green, blue) values in traditional two-digit hexadecimal format. Those integer values aren't directly available from the `Color` structure. One solution would be to calculate integer values of the color components within the ViewModel and expose them as properties. You could then format them using the `x2` formatting specification.

Another approach is more general: You can write a *binding value converter* as discussed in the later article, **Binding Value Converters**.

The next article, however, explores the **Binding Path** in more detail, and show how you can use it to reference sub-properties and items in collections.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms Binding Path

11/20/2018 • 3 minutes to read • [Edit Online](#)

In all the previous data-binding examples, the `Path` property of the `Binding` class (or the `Path` property of the `Binding` markup extension) has been set to a single property. It's actually possible to set `Path` to a *sub-property* (a property of a property), or to a member of a collection.

For example, suppose your page contains a `TimePicker`:

```
<TimePicker x:Name="timePicker">
```

The `Time` property of `TimePicker` is of type `TimeSpan`, but perhaps you want to create a data binding that references the `TotalSeconds` property of that `TimeSpan` value. Here's the data binding:

```
{Binding Source={x:Reference timePicker},  
        Path=Time.TotalSeconds}
```

The `Time` property is of type `TimeSpan`, which has a `TotalSeconds` property. The `Time` and `TotalSeconds` properties are simply connected with a period. The items in the `Path` string always refer to properties and not to the types of these properties.

That example and several others are shown in the **Path Variations** page:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:globe="clr-namespace:System.Globalization;assembly=mscorlib"
    x:Class="DataBindingDemos.PathVariationsPage"
    Title="Path Variations"
    x:Name="page">
<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="Large" />
            <Setter Property="HorizontalTextAlignment" Value="Center" />
            <Setter Property="VerticalOptions" Value="CenterAndExpand" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<StackLayout Margin="10, 0">
    <TimePicker x:Name="timePicker" />

    <Label Text="{Binding Source={x:Reference timePicker},
        Path=Time.TotalSeconds,
        StringFormat='{0} total seconds'}" />

    <Label Text="{Binding Source={x:Reference page},
        Path=Content.Children.Count,
        StringFormat='There are {0} children in this StackLayout'}" />

    <Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
        Path=DateTimeFormat.DayNames[3],
        StringFormat='The middle day of the week is {0}'}" />

    <Label>
        <Label.Text>
            <Binding Path="DateTimeFormat.DayNames[3]"
                StringFormat="The middle day of the week in France is {0}">
                <Binding.Source>
                    <globe:CultureInfo>
                        <x:Arguments>
                            <x:String>fr-FR</x:String>
                        </x:Arguments>
                    </globe:CultureInfo>
                </Binding.Source>
            </Binding>
        </Label.Text>
    </Label>

    <Label Text="{Binding Source={x:Reference page},
        Path=Content.Children[1].Text.Length,
        StringFormat='The second Label has {0} characters'}" />
</StackLayout>
</ContentPage>

```

In the second `Label`, the binding source is the page itself. The `Content` property is of type `StackLayout`, which has a `Children` property of type `IList<View>`, which has a `Count` property indicating the number of children.

## Paths with Indexers

The binding in the third `Label` in the **Path Variations** pages references the `CultureInfo` class in the `System.Globalization` namespace:

```

<Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
    Path=DateTimeFormat.DayNames[3],
    StringFormat='The middle day of the week is {0}'}" />

```

The source is set to the static `CultureInfo.CurrentCulture` property, which is an object of type `CultureInfo`. That class defines a property named `DateTimeFormat` of type `DateTimeFormatInfo` that contains a `DayNames` collection. The index selects the fourth item.

The fourth `Label` does something similar but for the culture associated with France. The `Source` property of the binding is set to `CultureInfo` object with a constructor:

```
<Label>
    <Label.Text>
        <Binding Path="DateTimeFormat.DayNames[3]"
            StringFormat="The middle day of the week in France is {0}">
            <Binding.Source>
                <globe:CultureInfo>
                    <x:Arguments>
                        <x:String>fr-FR</x:String>
                    </x:Arguments>
                </globe:CultureInfo>
            </Binding.Source>
        </Binding>
    </Label.Text>
</Label>
```

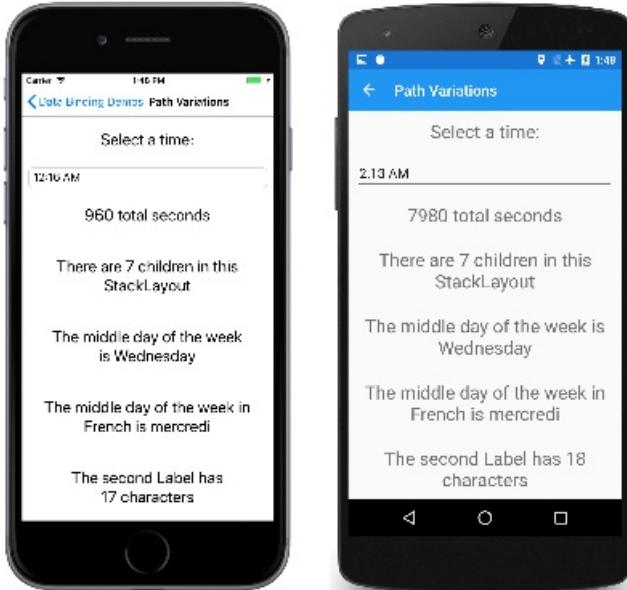
See [Passing Constructor Arguments](#) for more details on specifying constructor arguments in XAML.

Finally, the last example is similar to the second, except that it references one of the children of the `StackLayout`:

```
<Label Text="{Binding Source={x:Reference page},
    Path=Content.Children[1].Text.Length,
    StringFormat='The first Label has {0} characters'}" />
```

That child is a `Label`, which has a `Text` property of type `String`, which has a `Length` property. The first `Label` reports the `TimeSpan` set in the `TimePicker`, so when that text changes, the final `Label` changes as well.

Here's the program running:



## Debugging Complex Paths

Complex path definitions can be difficult to construct: You need to know the type of each sub-property or the type of items in the collection to correctly add the next sub-property, but the types themselves do not appear in the path. One good technique is to build up the path incrementally and look at the intermediate results. For that last

example, you could start with no `Path` definition at all:

```
<Label Text="{Binding Source={x:Reference page},  
StringFormat='{0}'}" />
```

That displays the type of the binding source, or `DataBindingDemos.PathVariationsPage`. You know `PathVariationsPage` derives from `ContentPage`, so it has a `Content` property:

```
<Label Text="{Binding Source={x:Reference page},  
Path=Content,  
StringFormat='{0}'}" />
```

The type of the `Content` property is now revealed to be `Xamarin.Forms.StackLayout`. Add the `Children` property to the `Path` and the type is `Xamarin.Forms.ElementCollection`1[Xamarin.Forms.View]`, which is a class internal to Xamarin.Forms, but obviously a collection type. Add an index to that and the type is `Xamarin.Forms.Label`. Continue in this way.

As Xamarin.Forms processes the binding path, it installs a `PropertyChanged` handler on any object in the path that implements the `INotifyPropertyChanged` interface. For example, the final binding reacts to a change in the first `Label` because the `Text` property changes.

If a property in the binding path does not implement `INotifyPropertyChanged`, any changes to that property will be ignored. Some changes could entirely invalidate the binding path, so you should use this technique only when the string of properties and sub-properties never become invalid.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms Binding Value Converters

7/12/2018 • 10 minutes to read • [Edit Online](#)

Data bindings usually transfer data from a source property to a target property, and in some cases from the target property to the source property. This transfer is straightforward when the source and target properties are of the same type, or when one type can be converted to the other type through an implicit conversion. When that is not the case, a type conversion must take place.

In the [String Formatting](#) article, you saw how you can use the `StringFormat` property of a data binding to convert any type into a string. For other types of conversions, you need to write some specialized code in a class that implements the `IValueConverter` interface. (The Universal Windows Platform contains a similar class named `IValueConverter` in the `Windows.UI.Xaml.Data` namespace, but this `IValueConverter` is in the `Xamarin.Forms` namespace.) Classes that implement `IValueConverter` are called *value converters*, but they are also often referred to as *binding converters* or *binding value converters*.

## The `IValueConverter` Interface

Suppose you want to define a data binding where the source property is of type `int` but the target property is a `bool`. You want this data binding to produce a `false` value when the integer source is equal to 0, and `true` otherwise.

You can do this with a class that implements the `IValueConverter` interface:

```
public class IntToBoolConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)value != 0;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (bool)value ? 1 : 0;
    }
}
```

You set an instance of this class to the `Converter` property of the `Binding` class or to the `Converter` property of the `Binding` markup extension. This class becomes part of the data binding.

The `Convert` method is called when data moves from the source to the target in `OneWay` or `TwoWay` bindings. The `value` parameter is the object or value from the data-binding source. The method must return a value of the type of the data-binding target. The method shown here casts the `value` parameter to an `int` and then compares it with 0 for a `bool` return value.

The `ConvertBack` method is called when data moves from the target to the source in `TwoWay` or `OneWayToSource` bindings. `ConvertBack` performs the opposite conversion: It assumes the `value` parameter is a `bool` from the target, and converts it to an `int` return value for the source.

If the data binding also includes a `StringFormat` setting, the value converter is invoked before the result is formatted as a string.

The [Enable Buttons](#) page in the [Data Binding Demos](#) sample demonstrates how to use this value converter in a data binding. The `IntToBoolConverter` is instantiated in the page's resource dictionary. It is then referenced with a

`StaticResource` markup extension to set the `Converter` property in two data bindings. It is very common to share data converters among multiple data bindings on the page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.EnableButtonsPage"
    Title="Enable Buttons">
<ContentPage.Resources>
    <ResourceDictionary>
        <local:IntToBoolConverter x:Key="intToBool" />
    </ResourceDictionary>
</ContentPage.Resources>

<StackLayout Padding="10, 0">
    <Entry x:Name="entry1"
        Text=""
        Placeholder="enter search term"
        VerticalOptions="CenterAndExpand" />

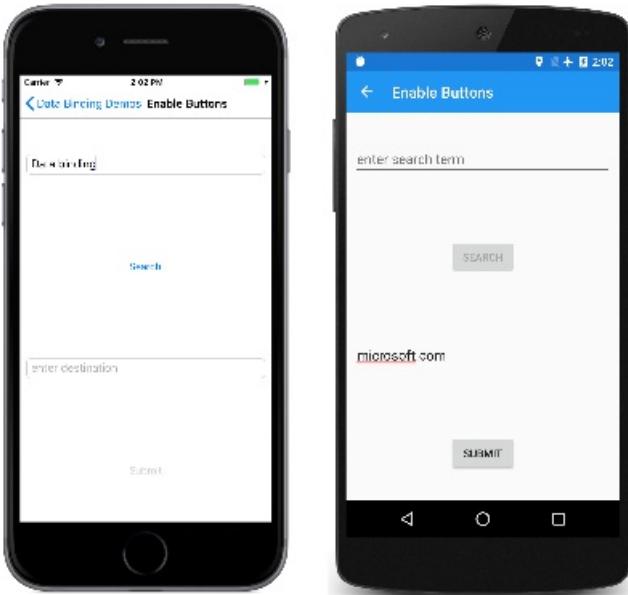
    <Button Text="Search"
        HorizontalOptions="Center"
        VerticalOptions="CenterAndExpand"
        IsEnabled="{Binding Source={x:Reference entry1},
            Path=Text.Length,
            Converter={StaticResource intToBool}}" />

    <Entry x:Name="entry2"
        Text=""
        Placeholder="enter destination"
        VerticalOptions="CenterAndExpand" />

    <Button Text="Submit"
        HorizontalOptions="Center"
        VerticalOptions="CenterAndExpand"
        IsEnabled="{Binding Source={x:Reference entry2},
            Path=Text.Length,
            Converter={StaticResource intToBool}}" />
</StackLayout>
</ContentPage>
```

If a value converter is used in multiple pages of your application, you can instantiate it in the resource dictionary in the **App.xaml** file.

The **Enable Buttons** page demonstrates a common need when a `Button` performs an operation based on text that the user types into an `Entry` view. If nothing has been typed into the `Entry`, the `Button` should be disabled. Each `Button` contains a data binding on its `.IsEnabled` property. The data-binding source is the `Length` property of the `Text` property of the corresponding `Entry`. If that `Length` property is not 0, the value converter returns `true` and the `Button` is enabled:



Notice that the `Text` property in each `Entry` is initialized to an empty string. The `Text` property is `null` by default, and the data binding will not work in that case.

Some value converters are written specifically for particular applications, while others are generalized. If you know that a value converter will only be used in `OneWay` bindings, then the `ConvertBack` method can simply return `null`.

The `Convert` method shown above implicitly assumes that the `value` argument is of type `int` and the return value must be of type `bool`. Similarly, the `ConvertBack` method assumes that the `value` argument is of type `bool` and the return value is `int`. If that is not the case, a runtime exception will occur.

You can write value converters to be more generalized and to accept several different types of data. The `Convert` and `ConvertBack` methods can use the `as` or `is` operators with the `value` parameter, or can call `GetType` on that parameter to determine its type, and then do something appropriate. The expected type of each method's return value is given by the `targetType` parameter. Sometimes, value converters are used with data bindings of different target types; the value converter can use the `targetType` argument to perform a conversion for the correct type.

If the conversion being performed is different for different cultures, use the `culture` parameter for this purpose. The `parameter` argument to `Convert` and `ConvertBack` is discussed later in this article.

## Binding Converter Properties

Value converter classes can have properties and generic parameters. This particular value converter converts a `bool` from the source to an object of type `T` for the target:

```

public class BoolToObjectConverter<T> : IValueConverter
{
    public T TrueObject { set; get; }

    public T FalseObject { set; get; }

    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (bool)value ? TrueObject : FalseObject;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return ((T)value).Equals(TrueObject);
    }
}

```

The **Switch Indicators** page demonstrates how it can be used to display the value of a `Switch` view. Although it's common to instantiate value converters as resources in a resource dictionary, this page demonstrates an alternative: Each value converter is instantiated between `Binding.Converter` property-element tags. The `x:TypeArguments` indicates the generic argument, and `TrueObject` and `FalseObject` are both set to objects of that type:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.SwitchIndicatorsPage"
    Title="Switch Indicators">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="18" />
                <Setter Property="VerticalOptions" Value="Center" />
            </Style>

            <Style TargetType="Switch">
                <Setter Property="VerticalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Padding="10, 0">
        <StackLayout Orientation="Horizontal"
            VerticalOptions="CenterAndExpand">
            <Label Text="Subscribe?" />
            <Switch x:Name="switch1" />
            <Label>
                <Label.Text>
                    <Binding Source="{x:Reference switch1}"
                        Path="IsToggled">
                        <Binding.Converter>
                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                TrueObject="Of course!">
                                <Binding.Converter>
                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                        TrueObject="Of course!">
                                        <Binding.Converter>
                                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                TrueObject="Of course!">
                                                <Binding.Converter>
                                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                        TrueObject="Of course!">
                                                        <Binding.Converter>
                                                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                TrueObject="Of course!">
                                                                <Binding.Converter>
                                                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                        TrueObject="Of course!">
                                                                        <Binding.Converter>
                                                                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                TrueObject="Of course!">
                                                                                <Binding.Converter>
                                                                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                        TrueObject="Of course!">
                                                                                        <Binding.Converter>
                                                                                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                                TrueObject="Of course!">
                                                                                                <Binding.Converter>
                                                                                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                                        TrueObject="Of course!">
                                                                                                        <Binding.Converter>
                                                                                                            <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                                                TrueObject="Of course!">
                                                                                                                <Binding.Converter>
                                                                                                                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                                                                                        TrueObject="Of course!">
                                                        </Binding.Converter>
                                                    </Binding.Converter>
                                                </Binding.Converter>
                                            </Binding.Converter>
                                        </Binding.Converter>
                                    </Binding.Converter>
                                </Binding.Converter>
                            </Binding.Converter>
                        </Binding.Converter>
                    </Binding>
                </Label.Text>
            </Label>
        </StackLayout>

        <StackLayout Orientation="Horizontal"
            VerticalOptions="CenterAndExpand">
            <Label Text="Allow popups?" />
            <Switch x:Name="switch2" />
            <Label>

```

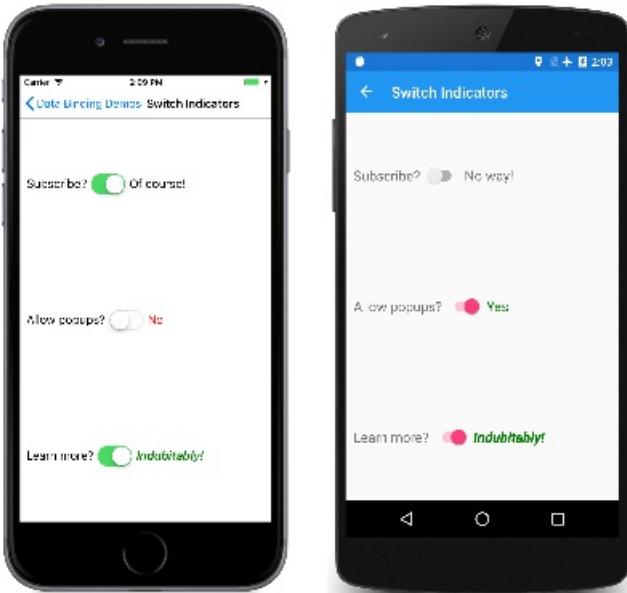
```

<Label>
    <Label.Text>
        <Binding Source="{x:Reference switch2}"
            Path="IsToggled">
            <Binding.Converter>
                <local:BoolToObjectConverter x:TypeArguments="x:String"
                    TrueObject="Yes"
                    FalseObject="No" />
            </Binding.Converter>
        </Binding>
    </Label.Text>
    <Label.TextColor>
        <Binding Source="{x:Reference switch2}"
            Path="IsToggled">
            <Binding.Converter>
                <local:BoolToObjectConverter x:TypeArguments="Color"
                    TrueObject="Green"
                    FalseObject="Red" />
            </Binding.Converter>
        </Binding>
    </Label.TextColor>
</Label>
</StackLayout>

<StackLayout Orientation="Horizontal"
    VerticalOptions="CenterAndExpand">
    <Label Text="Learn more?" />
    <Switch x:Name="switch3" />
    <Label FontSize="18"
        VerticalOptions="Center">
        <Label.Style>
            <Binding Source="{x:Reference switch3}"
                Path="IsToggled">
                <Binding.Converter>
                    <local:BoolToObjectConverter x:TypeArguments="Style">
                        <local:BoolToObjectConverter.TrueObject>
                            <Style TargetType="Label">
                                <Setter Property="Text" Value="Indubitably!" />
                                <Setter Property="FontAttributes" Value="Italic, Bold" />
                                <Setter Property="TextColor" Value="Green" />
                            </Style>
                        </local:BoolToObjectConverter.TrueObject>
                    </local:BoolToObjectConverter.FalseObject>
                        <Style TargetType="Label">
                            <Setter Property="Text" Value="Maybe later" />
                            <Setter Property="FontAttributes" Value="None" />
                            <Setter Property="TextColor" Value="Red" />
                        </Style>
                    </local:BoolToObjectConverter.FalseObject>
                </local:BoolToObjectConverter>
            </Binding.Converter>
        </Binding>
    </Label.Style>
</Label>
</StackLayout>
</StackLayout>
</ContentPage>

```

In the last of the three `Switch` and `Label` pairs, the generic argument is set to `Style`, and entire `Style` objects are provided for the values of `TrueObject` and `FalseObject`. These override the implicit style for `Label` set in the resource dictionary, so the properties in that style are explicitly assigned to the `Label`. Toggling the `Switch` causes the corresponding `Label` to reflect the change:



It's also possible to use [Triggers](#) to implement similar changes in the user-interface based on other views.

## Binding Converter Parameters

The `Binding` class defines a `ConverterParameter` property, and the `Binding` markup extension also defines a `ConverterParameter` property. If this property is set, then the value is passed to the `Convert` and `ConvertBack` methods as the `parameter` argument. Even if the instance of the value converter is shared among several data bindings, the `ConverterParameter` can be different to perform somewhat different conversions.

The use of `ConverterParameter` is demonstrated with a color-selection program. In this case, the `RgbColorViewModel` has three properties of type `double` named `Red`, `Green`, and `Blue` that it uses to construct a `Color` value:

```
public class RgbColorViewModel : INotifyPropertyChanged
{
    Color color;
    string name;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Red
    {
        set
        {
            if (color.R != value)
            {
                Color = new Color(value, color.G, color.B);
            }
        }
        get
        {
            return color.R;
        }
    }

    public double Green
    {
        set
        {
            if (color.G != value)
            {
                Color = new Color(color.R, value, color.B);
            }
        }
    }

    public double Blue
    {
        set
        {
            if (color.B != value)
            {
                Color = new Color(color.R, color.G, value);
            }
        }
    }
}
```

```

        }
        get
        {
            return color.G;
        }
    }

    public double Blue
    {
        set
        {
            if (color.B != value)
            {
                Color = new Color(color.R, color.G, value);
            }
        }
        get
        {
            return color.B;
        }
    }

    public Color Color
    {
        set
        {
            if (color != value)
            {
                color = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Red"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Green"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Blue"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));

                Name = NamedColor.GetNearestColorName(color);
            }
        }
        get
        {
            return color;
        }
    }

    public string Name
    {
        private set
        {
            if (name != value)
            {
                name = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
            }
        }
        get
        {
            return name;
        }
    }
}

```

The `Red`, `Green`, and `Blue` properties range between 0 and 1. However, you might prefer that the components be displayed as two-digit hexadecimal values.

To display these as hexadecimal values in XAML, they must be multiplied by 255, converted to an integer, and then formatted with a specification of "X2" in the `StringFormat` property. The first two tasks (multiplying by 255 and converting to an integer) can be handled by the value converter. To make the value converter as generalized as

possible, the multiplication factor can be specified with the `ConverterParameter` property, which means that it enters the `Convert` and `ConvertBack` methods as the `parameter` argument:

```
public class DoubleToIntConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)Math.Round((double)value * GetParameter(parameter));
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)value / GetParameter(parameter);
    }

    double GetParameter(object parameter)
    {
        if (parameter is double)
            return (double)parameter;

        else if (parameter is int)
            return (int)parameter;

        else if (parameter is string)
            return double.Parse((string)parameter);

        return 1;
    }
}
```

The `Convert` converts from a `double` to `int` while multiplying by the `parameter` value; the `ConvertBack` divides the integer `value` argument by `parameter` and returns a `double` result. (In the program shown below, the value converter is used only in connection with string formatting, so `ConvertBack` is not used.)

The type of the `parameter` argument is likely to be different depending on whether the data binding is defined in code or XAML. If the `ConverterParameter` property of `Binding` is set in code, it's likely to be set to a numeric value:

```
binding.ConverterParameter = 255;
```

The `ConverterParameter` property is of type `Object`, so the C# compiler interprets the literal 255 as an integer, and sets the property to that value.

In XAML, however, the `ConverterParameter` is likely to be set like this:

```
<Label Text="{Binding Red,
           Converter={StaticResource doubleToInt},
           ConverterParameter=255,
           StringFormat='Red = {0:X2}'}" />
```

The 255 looks like a number, but because `ConverterParameter` is of type `Object`, the XAML parser treats the 255 as a string.

For that reason, the value converter shown above includes a separate `GetParameter` method that handles cases for `parameter` being of type `double`, `int`, or `string`.

The **RGB Color Selector** page instantiates `DoubleToIntConverter` in its resource dictionary following the definition of two implicit styles:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.RgbColorSelectorPage"
    Title="RGB Color Selector">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <local:DoubleToIntConverter x:Key="doubleToInt" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <StackLayout.BindingContext>
            <local:RgbColorViewModel Color="Gray" />
        </StackLayout.BindingContext>

        <BoxView Color="{Binding Color}"
            VerticalOptions="FillAndExpand" />

        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />

            <Slider Value="{Binding Red}" />
            <Label Text="{Binding Red,
                Converter={StaticResource doubleToInt},
                ConverterParameter=255,
                StringFormat='Red = {0:X2}'}" />

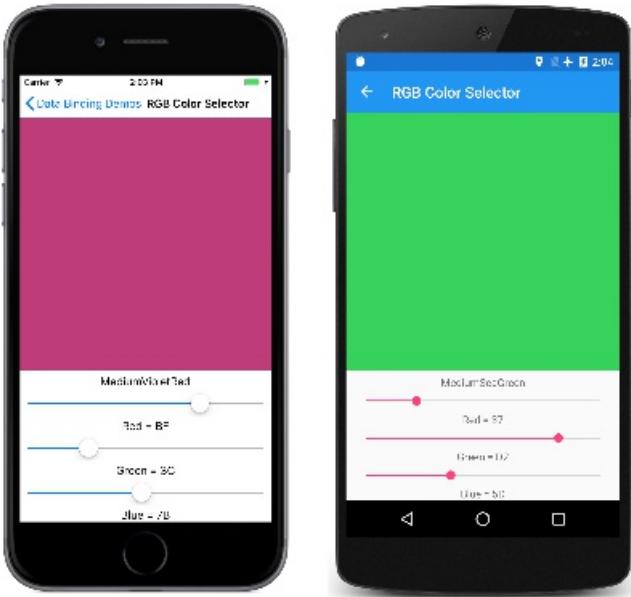
            <Slider Value="{Binding Green}" />
            <Label Text="{Binding Green,
                Converter={StaticResource doubleToInt},
                ConverterParameter=255,
                StringFormat='Green = {0:X2}'}" />

            <Slider Value="{Binding Blue}" />
            <Label>
                <Label.Text>
                    <Binding Path="Blue"
                        StringFormat="Blue = {0:X2}"
                        Converter="{StaticResource doubleToInt}">
                        <Binding.ConverterParameter>
                            <x:Double>255</x:Double>
                        </Binding.ConverterParameter>
                    </Binding>
                </Label.Text>
            </Label>
        </StackLayout>
    </StackLayout>
</ContentPage>

```

The values of the `Red` and `Green` properties are displayed with a `Binding` markup extension. The `Blue` property, however, instantiates the `Binding` class to demonstrate how an explicit `double` value can be set to `ConverterParameter` property.

Here's the result:



## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms Binding Fallbacks

11/20/2018 • 3 minutes to read • [Edit Online](#)

Sometimes data bindings fail, because the binding source can't be resolved, or because the binding succeeds but returns a `null` value. While these scenarios can be handled with value converters, or other additional code, data bindings can be made more robust by defining fallback values to use if the binding process fails. This can be accomplished by defining the `FallbackValue` and `TargetNullValue` properties in a binding expression. Because these properties reside in the `BindingBase` class, they can be used with bindings, compiled bindings, and with the `Binding` markup extension.

## NOTE

Use of the `FallbackValue` and `TargetNullValue` properties in a binding expression is optional.

## Defining a fallback value

The `FallbackValue` property allows a fallback value to be defined that will be used when the binding source can't be resolved. A common scenario for setting this property is when binding to source properties that might not exist on all objects in a bound collection of heterogeneous types.

The [MonkeyDetail](#) page illustrates setting the `FallbackValue` property:

```
<Label Text="{Binding Population, FallbackValue='Population size unknown'}"  
... />
```

The binding on the `Label` defines a `FallbackValue` value that will be set on the target if the binding source can't be resolved. Therefore, the value defined by the `FallbackValue` property will be displayed if the `Population` property doesn't exist on the bound object. Notice that here the `FallbackValue` property value is delimited by single-quote (apostrophe) characters.

Rather than defining `FallbackValue` property values inline, it's recommended to define them as resources in a `ResourceDictionary`. The advantage of this approach is that such values are defined once in a single location, and are more easily localizable. The resources can then be retrieved using the `StaticResource` markup extension:

```
<Label Text="{Binding Population, FallbackValue={StaticResource populationUnknown}}"  
... />
```

## NOTE

It's not possible to set the `FallbackValue` property with a binding expression.

Here's the program running:



When the `FallbackValue` property isn't set in a binding expression and the binding path or part of the path isn't resolved, `BindableProperty.DefaultValue` is set on the target. However, when the `FallbackValue` property is set and the binding path or part of the path isn't resolved, the value of the `FallbackValue` value property is set on the target. Therefore, on the **MonkeyDetail** page the `Label` displays "Population size unknown" because the bound object lacks a `Population` property.

#### IMPORTANT

A defined value converter is not executed in a binding expression when the `FallbackValue` property is set.

## Defining a null replacement value

The `TargetNullValue` property allows a replacement value to be defined that will be used when the binding source is resolved, but the value is `null`. A common scenario for setting this property is when binding to source properties that might be `null` in a bound collection.

The **Monkeys** page illustrates setting the `TargetNullValue` property:

```
<ListView ItemsSource="{Binding Monkeys}"
    ...>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <Grid>
                    ...
                    <Image Source="{Binding ImageUrl,
TargetNullValue='https://upload.wikimedia.org/wikipedia/commons/2/20/Point_d_interrogation.jpg'}"
                        ... />
                    ...
                    <Label Text="{Binding Location, TargetNullValue='Location unknown'}"
                        ... />
                </Grid>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

The bindings on the `Image` and `Label` both define `TargetNullValue` values that will be applied if the binding path returns `null`. Therefore, the values defined by the `TargetNullValue` properties will be displayed for any objects in the collection where the `ImageUrl` and `Location` properties are not defined. Notice that here the `TargetNullValue` property values are delimited by single-quote (apostrophe) characters.

Rather than defining `TargetNullValue` property values inline, it's recommended to define them as resources in a `ResourceDictionary`. The advantage of this approach is that such values are defined once in a single location, and are more easily localizable. The resources can then be retrieved using the `StaticResource` markup extension:

```

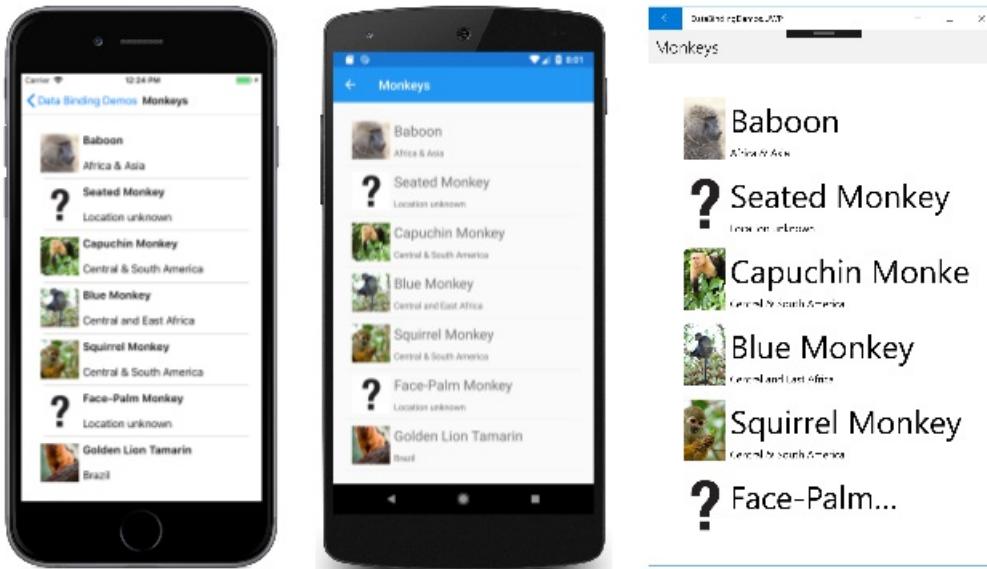
<Image Source="{Binding ImageUrl, TargetNullValue={StaticResource fallbackImageUrl}}"
... />
<Label Text="{Binding Location, TargetNullValue={StaticResource locationUnknown}}"
... />

```

#### NOTE

It's not possible to set the `TargetNullValue` property with a binding expression.

Here's the program running:



When the `TargetNullValue` property isn't set in a binding expression, a source value of `null` will be converted if a value converter is defined, formatted if a `StringFormat` is defined, and the result is then set on the target. However, when the `TargetNullValue` property is set, a source value of `null` will be converted if a value converter is defined, and if it's still `null` after the conversion, the value of the `TargetNullValue` property is set on the target.

#### IMPORTANT

String formatting is not applied in a binding expression when the `TargetNullValue` property is set.

## Related Links

- [Data Binding Demos \(sample\)](#)

# The Xamarin.Forms Command Interface

10/31/2018 • 17 minutes to read • [Edit Online](#)

In the Model-View-ViewModel (MVVM) architecture, data bindings are defined between properties in the ViewModel, which is generally a class that derives from `INotifyPropertyChanged`, and properties in the View, which is generally the XAML file. Sometimes an application has needs that go beyond these property bindings by requiring the user to initiate commands that affect something in the ViewModel. These commands are generally signaled by button clicks or finger taps, and traditionally they are processed in the code-behind file in a handler for the `Clicked` event of the `Button` or the `Tapped` event of a `TapGestureRecognizer`.

The commanding interface provides an alternative approach to implementing commands that is much better suited to the MVVM architecture. The ViewModel itself can contain commands, which are methods that are executed in reaction to a specific activity in the View such as a `Button` click. Data bindings are defined between these commands and the `Button`.

To allow a data binding between a `Button` and a ViewModel, the `Button` defines two properties:

- `Command` of type `System.Windows.Input.ICommand`
- `CommandParameter` of type `Object`

To use the command interface, you define a data binding that targets the `Command` property of the `Button` where the source is a property in the ViewModel of type `ICommand`. The ViewModel contains code associated with that `ICommand` property that is executed when the button is clicked. You can set `CommandParameter` to arbitrary data to distinguish between multiple buttons if they are all bound to the same `ICommand` property in the ViewModel.

The `Command` and `CommandParameter` properties are also defined by the following classes:

- `MenuItem` and hence, `ToolbarItem`, which derives from `MenuItem`
- `TextCell` and hence, `ImageCell`, which derives from `TextCell`
- `TapGestureRecognizer`

`SearchBar` defines a `SearchCommand` property of type `ICommand` and a `SearchCommandParameter` property. The `RefreshCommand` property of `ListView` is also of type `ICommand`.

All these commands can be handled within a ViewModel in a manner that doesn't depend on the particular user-interface object in the View.

## The ICommand Interface

The `System.Windows.Input.ICommand` interface is not part of Xamarin.Forms. It is defined instead in the `System.Windows.Input` namespace, and consists of two methods and one event:

```
public interface ICommand
{
    public void Execute (Object parameter);

    public bool CanExecute (Object parameter);

    public event EventHandler CanExecuteChanged;
}
```

To use the command interface, your ViewModel contains properties of type `ICommand`:

```
public ICommand MyCommand { private set; get; }
```

The ViewModel must also reference a class that implements the `ICommand` interface. This class will be described shortly. In the View, the `Command` property of a `Button` is bound to that property:

```
<Button Text="Execute command"  
       Command="{Binding MyCommand}" />
```

When the user presses the `Button`, the `Button` calls the `Execute` method in the `ICommand` object bound to its `Command` property. That's the simplest part of the commanding interface.

The `CanExecute` method is more complex. When the binding is first defined on the `Command` property of the `Button`, and when the data binding changes in some way, the `Button` calls the `CanExecute` method in the `ICommand` object. If `CanExecute` returns `false`, then the `Button` disables itself. This indicates that the particular command is currently unavailable or invalid.

The `Button` also attaches a handler on the `CanExecuteChanged` event of `ICommand`. The event is fired from within the ViewModel. When that event is fired, the `Button` calls `CanExecute` again. The `Button` enables itself if `CanExecute` returns `true` and disables itself if `CanExecute` returns `false`.

#### IMPORTANT

Do not use the `IsEnabled` property of `Button` if you're using the command interface.

## The Command Class

When your ViewModel defines a property of type `ICommand`, the ViewModel must also contain or reference a class that implements the `ICommand` interface. This class must contain or reference the `Execute` and `CanExecute` methods, and fire the `CanExecuteChanged` event whenever the `CanExecute` method might return a different value.

You can write such a class yourself, or you can use a class that someone else has written. Because `ICommand` is part of Microsoft Windows, it has been used for years with Windows MVVM applications. Using a Windows class that implements `ICommand` allows you to share your ViewModels between Windows applications and Xamarin.Forms applications.

If sharing ViewModels between Windows and Xamarin.Forms is not a concern, then you can use the `Command` or `Command<T>` class included in Xamarin.Forms to implement the `ICommand` interface. These classes allow you to specify the bodies of the `Execute` and `CanExecute` methods in class constructors. Use `Command<T>` when you use the `CommandParameter` property to distinguish between multiple views bound to the same `ICommand` property, and the simpler `Command` class when that isn't a requirement.

## Basic Commanding

The **Person Entry** page in the [Data Binding Demos](#) program demonstrates some simple commands implemented in a ViewModel.

The `PersonViewModel` defines three properties named `Name`, `Age`, and `Skills` that define a person. This class does *not* contain any `ICommand` properties:

```

public class PersonViewModel : INotifyPropertyChanged
{
    string name;
    double age;
    string skills;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Name
    {
        set { SetProperty(ref name, value); }
        get { return name; }
    }

    public double Age
    {
        set { SetProperty(ref age, value); }
        get { return age; }
    }

    public string Skills
    {
        set { SetProperty(ref skills, value); }
        get { return skills; }
    }

    public override string ToString()
    {
        return Name + ", " + Age;
    }

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

The `PersonCollectionViewModel` shown below creates new objects of type `PersonViewModel` and allows the user to fill in the data. For that purpose, the class defines properties `IsEditing` of type `bool` and `PersonEdit` of type `PersonViewModel`. In addition, the class defines three properties of type `ICommand` and a property named `Persons` of type `IList<PersonViewModel>`:

```

public class PersonCollectionViewModel : INotifyPropertyChanged
{
    PersonViewModel personEdit;
    bool isEditing;

    public event PropertyChangedEventHandler PropertyChanged;

    ...

    public bool IsEditing
    {
        private set { SetProperty(ref isEditing, value); }
        get { return isEditing; }
    }

    public PersonViewModel PersonEdit
    {
        set { SetProperty(ref personEdit, value); }
        get { return personEdit; }
    }

    public ICommand NewCommand { private set; get; }

    public ICommand SubmitCommand { private set; get; }

    public ICommand CancelCommand { private set; get; }

    public IList<PersonViewModel> Persons { get; } = new ObservableCollection<PersonViewModel>();

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

This abbreviated listing does not include the class's constructor, which is where the three properties of type `ICommand` are defined, which will be shown shortly. Notice that changes to the three properties of type `ICommand` and the `Persons` property do not result in `PropertyChanged` events being fired. These properties are all set when the class is first created and do not change thereafter.

Before examining the constructor of the `PersonCollectionViewModel` class, let's look at the XAML file for the **Person Entry** program. This contains a `Grid` with its `BindingContext` property set to the `PersonCollectionViewModel`. The `Grid` contains a `Button` with the text **New** with its `Command` property bound to the `NewCommand` property in the ViewModel, an entry form with properties bound to the `IsEditing` property, as well as properties of `PersonViewModel`, and two more buttons bound to the `SubmitCommand` and `CancelCommand` properties of the ViewModel. The final `ListView` displays the collection of persons already entered:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.PersonEntryPage"
    Title="Person Entry">
    <Grid Margin="10">

```

```

<Grid.BindingContext>
    <local:PersonCollectionViewModel />
</Grid.BindingContext>

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<!-- New Button -->
<Button Text="New"
        Grid.Row="0"
        Command="{Binding NewCommand}"
        HorizontalOptions="Start" />

<!-- Entry Form -->
<Grid Grid.Row="1"
      IsEnabled="{Binding IsEditing}">

    <Grid BindingContext="{Binding PersonEdit}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Label Text="Name: " Grid.Row="0" Grid.Column="0" />
        <Entry Text="{Binding Name}"
              Grid.Row="0" Grid.Column="1" />

        <Label Text="Age: " Grid.Row="1" Grid.Column="0" />
        <StackLayout Orientation="Horizontal"
                    Grid.Row="1" Grid.Column="1">
            <Stepper Value="{Binding Age}"
                     Maximum="100" />
            <Label Text="{Binding Age, StringFormat='{0} years old'}"
                  VerticalOptions="Center" />
        </StackLayout>

        <Label Text="Skills: " Grid.Row="2" Grid.Column="0" />
        <Entry Text="{Binding Skills}"
              Grid.Row="2" Grid.Column="1" />

    </Grid>
</Grid>

<!-- Submit and Cancel Buttons -->
<Grid Grid.Row="2">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Text="Submit"
           Grid.Column="0"
           Command="{Binding SubmitCommand}"
           VerticalOptions="CenterAndExpand" />

    <Button Text="Cancel"
           Grid.Column="1"
           Command="{Binding CancelCommand}"
           VerticalOptions="CenterAndExpand" />

```

```

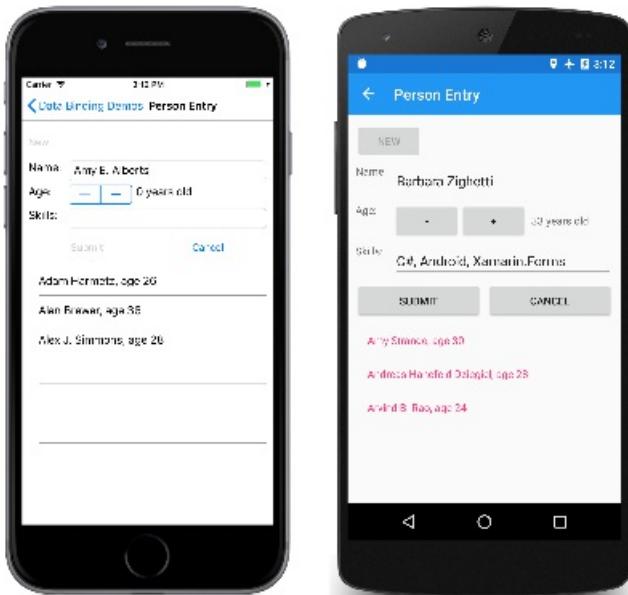
        </Grid>

        <!-- List of Persons -->
        <ListView Grid.Row="3"
                  ItemsSource="{Binding Persons}" />
    </Grid>
</ContentPage>

```

Here's how it works: The user first presses the **New** button. This enables the entry form but disables the **New** button. The user then enters a name, age, and skills. At any time during the editing, the user can press the **Cancel** button to start over. Only when a name and a valid age have been entered is the **Submit** button enabled. Pressing this **Submit** button transfers the person to the collection displayed by the `ListView`. After either the **Cancel** or **Submit** button is pressed, the entry form is cleared and the **New** button is enabled again.

The iOS screen at the left shows the layout before a valid age is entered. The Android and UWP screens show the **Submit** button enabled after an age has been set:



The program does not have any facility for editing existing entries, and does not save the entries when you navigate away from the page.

All the logic for the **New**, **Submit**, and **Cancel** buttons is handled in `PersonCollectionViewModel` through definitions of the `NewCommand`, `SubmitCommand`, and `CancelCommand` properties. The constructor of the `PersonCollectionViewModel` sets these three properties to objects of type `Command`.

A [constructor](#) of the `Command` class allows you to pass arguments of type `Action` and `Func<bool>` corresponding to the `Execute` and `CanExecute` methods. It's easiest to define these actions and functions as lambda functions right in the `Command` constructor. Here is the definition of the `Command` object for the `NewCommand` property:

```

public class PersonCollectionViewModel : INotifyPropertyChanged
{
    ...

    public PersonCollectionViewModel()
    {
        NewCommand = new Command(
            execute: () =>
            {
                PersonEdit = new PersonViewModel();
                PersonEdit.PropertyChanged += OnPersonEditPropertyChanged;
                IsEditing = true;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return !IsEditing;
            });
        ...
    }

    void OnPersonEditPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        (SubmitCommand as Command).ChangeCanExecute();
    }

    void RefreshCanExecutes()
    {
        (NewCommand as Command).ChangeCanExecute();
        (SubmitCommand as Command).ChangeCanExecute();
        (CancelCommand as Command).ChangeCanExecute();
    }
    ...
}

```

When the user clicks the **New** button, the `execute` function passed to the `Command` constructor is executed. This creates a new `PersonViewModel` object, sets a handler on that object's `PropertyChanged` event, sets `IsEditing` to `true`, and calls the `RefreshCanExecutes` method defined after the constructor.

Besides implementing the `ICommand` interface, the `Command` class also defines a method named `ChangeCanExecute`. Your ViewModel should call `ChangeCanExecute` for an `ICommand` property whenever anything happens that might change the return value of the `CanExecute` method. A call to `ChangeCanExecute` causes the `Command` class to fire the `CanExecuteChanged` method. The `Button` has attached a handler for that event and responds by calling `CanExecute` again, and then enabling itself based on the return value of that method.

When the `execute` method of `NewCommand` calls `RefreshCanExecutes`, the `NewCommand` property gets a call to `ChangeCanExecute`, and the `Button` calls the `canExecute` method, which now returns `false` because the `IsEditing` property is now `true`.

The `PropertyChanged` handler for the new `PersonViewModel` object calls the `ChangeCanExecute` method of `SubmitCommand`. Here's how that command property is implemented:

```

public class PersonCollectionViewModel : INotifyPropertyChanged
{
    ...

    public PersonCollectionViewModel()
    {
        ...

        SubmitCommand = new Command(
            execute: () =>
            {
                Persons.Add(PersonEdit);
                PersonEdit.PropertyChanged -= OnPersonEditPropertyChanged;
                PersonEdit = null;
                IsEditing = false;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return PersonEdit != null &&
                    PersonEdit.Name != null &&
                    PersonEdit.Name.Length > 1 &&
                    PersonEdit.Age > 0;
            });
        ...
    }

    ...
}

```

The `canExecute` function for `SubmitCommand` is called every time there's a property changed in the `PersonViewModel` object being edited. It returns `true` only when the `Name` property is at least one character long, and `Age` is greater than 0. At that time, the **Submit** button becomes enabled.

The `execute` function for **Submit** removes the property-changed handler from the `PersonViewModel`, adds the object to the `Persons` collection, and returns everything to initial conditions.

The `execute` function for the **Cancel** button does everything that the **Submit** button does except add the object to the collection:

```

public class PersonCollectionViewModel : INotifyPropertyChanged
{
    ...

    public PersonCollectionViewModel()
    {
        ...

        CancelCommand = new Command(
            execute: () =>
            {
                PersonEdit.PropertyChanged -= OnPersonEditPropertyChanged;
                PersonEdit = null;
                IsEditing = false;
                RefreshCanExecute();
            },
            canExecute: () =>
            {
                return IsEditing;
            });
    }

    ...
}

```

The `canExecute` method returns `true` at any time a `PersonViewModel` is being edited.

These techniques could be adapted to more complex scenarios: A property in `PersonCollectionViewModel` could be bound to the `SelectedItem` property of the `ListView` for editing existing items, and a **Delete** button could be added to delete those items.

It isn't necessary to define the `execute` and `canExecute` methods as lambda functions. You can write them as regular private methods in the ViewModel and reference them in the `Command` constructors. However, this approach does tend to result in a lot of methods that are referenced only once in the ViewModel.

## Using Command Parameters

It is sometimes convenient for one or more buttons (or other user-interface objects) to share the same `ICommand` property in the ViewModel. In this case, you use the `CommandParameter` property to distinguish between the buttons.

You can continue to use the `Command` class for these shared `ICommand` properties. The class defines an [alternative constructor](#) that accepts `execute` and `canExecute` methods with parameters of type `object`. This is how the `CommandParameter` is passed to these methods.

However, when using `CommandParameter`, it's easiest to use the generic `Command<T>` class to specify the type of the object set to `CommandParameter`. The `execute` and `canExecute` methods that you specify have parameters of that type.

The **Decimal Keyboard** page illustrates this technique by showing how to implement a keypad for entering decimal numbers. The `BindingContext` for the `Grid` is a `DecimalKeypadViewModel`. The `Entry` property of this ViewModel is bound to the `Text` property of a `Label`. All the `Button` objects are bound to various commands in the ViewModel: `ClearCommand`, `BackspaceCommand`, and `DigitCommand`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"

```

```
x:Class="DataBindingDemos.DecimalKeypadPage"
Title="Decimal Keyboard"

<Grid WidthRequest="240"
      HeightRequest="480"
      ColumnSpacing="2"
      RowSpacing="2"
      HorizontalOptions="Center"
      VerticalOptions="Center">

    <Grid.BindingContext>
        <local:DecimalKeypadViewModel />
    </Grid.BindingContext>

    <Grid.Resources>
        <ResourceDictionary>
            <Style TargetType="Button">
                <Setter Property="FontSize" Value="32" />
                <Setter Property="BorderWidth" Value="1" />
                <Setter Property="BorderColor" Value="Black" />
            </Style>
        </ResourceDictionary>
    </Grid.Resources>

    <Label Text="{Binding Entry}"
          Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3"
          FontSize="32"
          LineBreakMode="HeadTruncation"
          VerticalTextAlignment="Center"
          HorizontalTextAlignment="End" />

    <Button Text="CLEAR"
          Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2"
          Command="{Binding ClearCommand}" />

    <Button Text="⌫"
          Grid.Row="1" Grid.Column="2"
          Command="{Binding BackspaceCommand}" />

    <Button Text="7"
          Grid.Row="2" Grid.Column="0"
          Command="{Binding DigitCommand}"
          CommandParameter="7" />

    <Button Text="8"
          Grid.Row="2" Grid.Column="1"
          Command="{Binding DigitCommand}"
          CommandParameter="8" />

    <Button Text="9"
          Grid.Row="2" Grid.Column="2"
          Command="{Binding DigitCommand}"
          CommandParameter="9" />

    <Button Text="4"
          Grid.Row="3" Grid.Column="0"
          Command="{Binding DigitCommand}"
          CommandParameter="4" />

    <Button Text="5"
          Grid.Row="3" Grid.Column="1"
          Command="{Binding DigitCommand}"
          CommandParameter="5" />

    <Button Text="6"
          Grid.Row="3" Grid.Column="2"
          Command="{Binding DigitCommand}"
          CommandParameter="6" />
```

```

<Button Text="1"
    Grid.Row="4" Grid.Column="0"
    Command="{Binding DigitCommand}"
    CommandParameter="1" />

<Button Text="2"
    Grid.Row="4" Grid.Column="1"
    Command="{Binding DigitCommand}"
    CommandParameter="2" />

<Button Text="3"
    Grid.Row="4" Grid.Column="2"
    Command="{Binding DigitCommand}"
    CommandParameter="3" />

<Button Text="0"
    Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2"
    Command="{Binding DigitCommand}"
    CommandParameter="0" />

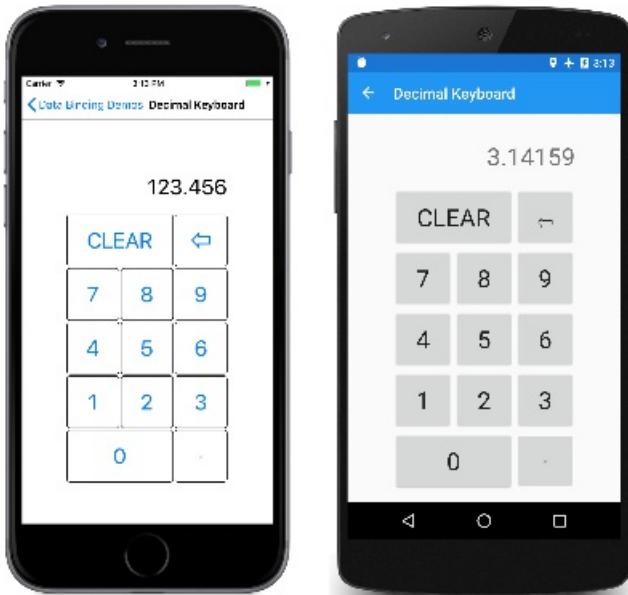
<Button Text=" ."
    Grid.Row="5" Grid.Column="2"
    Command="{Binding DigitCommand}"
    CommandParameter="." />

</Grid>
</ContentPage>

```

The 11 buttons for the 10 digits and the decimal point share a binding to `DigitCommand`. The `CommandParameter` distinguishes between these buttons. The value set to `CommandParameter` is generally the same as the text displayed by the button except for the decimal point, which for purposes of clarity is displayed with a middle dot character.

Here's the program in action:



Notice that the button for the decimal point in all three screenshots is disabled because the entered number already contains a decimal point.

The `DecimalKeypadViewModel` defines an `Entry` property of type `string` (which is the only property that triggers a `PropertyChanged` event) and three properties of type `ICommand`:

```

public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    string entry = "0";

    public event PropertyChangedEventHandler PropertyChanged;

    ...

    public string Entry
    {
        private set
        {
            if (entry != value)
            {
                entry = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Entry"));
            }
        }
        get
        {
            return entry;
        }
    }

    public ICommand ClearCommand { private set; get; }

    public ICommand BackspaceCommand { private set; get; }

    public ICommand DigitCommand { private set; get; }
}

```

The button corresponding to the `ClearCommand` is always enabled and simply sets the entry back to "0":

```

public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    ...

    public DecimalKeypadViewModel()
    {
        ClearCommand = new Command(
            execute: () =>
            {
                Entry = "0";
                RefreshCanExecutes();
            });
    }

    void RefreshCanExecutes()
    {
        ((Command)BackspaceCommand).ChangeCanExecute();
        ((Command)DigitCommand).ChangeCanExecute();
    }

    ...
}

}

```

Because the button is always enabled, it is not necessary to specify a `canExecute` argument in the `Command` constructor.

The logic for entering numbers and backspacing is a little tricky because if no digits have been entered, then the `Entry` property is the string "0". If the user types more zeroes, then the `Entry` still contains just one zero. If the user types any other digit, that digit replaces the zero. But if the user types a decimal point before any other digit, then `Entry` is the string "0".

The **Backspace** button is enabled only when the length of the entry is greater than 1, or if `Entry` is not equal to the string "0":

```
public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    ...
    public DecimalKeypadViewModel()
    {
        ...
        BackspaceCommand = new Command(
            execute: () =>
            {
                Entry = Entry.Substring(0, Entry.Length - 1);
                if (Entry == "")
                {
                    Entry = "0";
                }
                RefreshCanExecute();
            },
            canExecute: () =>
            {
                return Entry.Length > 1 || Entry != "0";
            });
        ...
    }
    ...
}
```

The logic for the `execute` function for the **Backspace** button ensures that the `Entry` is at least a string of "0".

The `DigitCommand` property is bound to 11 buttons, each of which identifies itself with the `CommandParameter` property. The `DigitCommand` could be set to an instance of the regular `Command` class, but it's easier to use the `Command<T>` generic class. When using the commanding interface with XAML, the `CommandParameter` properties are usually strings, and that's the type of the generic argument. The `execute` and `canExecute` functions then have arguments of type `string`:

```

public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    ...

    public DecimalKeypadViewModel()
    {
        ...

        DigitCommand = new Command<string>(
            execute: (string arg) =>
            {
                Entry += arg;
                if (Entry.StartsWith("0") && !Entry.StartsWith("0."))
                {
                    Entry = Entry.Substring(1);
                }
                RefreshCanExecutes();
            },
            canExecute: (string arg) =>
            {
                return !(arg == "." && Entry.Contains("."));
            });
    }

    ...
}

}

```

The `execute` method appends the string argument to the `Entry` property. However, if the result begins with a zero (but not a zero and a decimal point) then that initial zero must be removed using the `Substring` function.

The `canExecute` method returns `false` only if the argument is the decimal point (indicating that the decimal point is being pressed) and `Entry` already contains a decimal point.

All the `execute` methods call `RefreshCanExecutes`, which then calls `ChangeCanExecute` for both `DigitCommand` and `ClearCommand`. This ensures that the decimal point and backspace buttons are enabled or disabled based on the current sequence of entered digits.

## Adding Commands to Existing Views

If you'd like to use the commanding interface with views that don't support it, it's possible to use a `Xamarin.Forms` behavior that converts an event into a command. This is described in the article [Reusable EventToCommandBehavior](#).

## Asynchronous Commanding for Navigation Menus

Commanding is convenient for implementing navigation menus, such as that in the [Data Binding Demos](#) program itself. Here's part of `MainPage.xaml`:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.MainPage"
    Title="Data Binding Demos"
    Padding="10">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection Title="Basic Bindings">

                <TextCell Text="Basic Code Binding"
                    Detail="Define a data-binding in code"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:BasicCodeBindingPage}" />

                <TextCell Text="Basic XAML Binding"
                    Detail="Define a data-binding in XAML"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:BasicXamlBindingPage}" />

                <TextCell Text="Alternative Code Binding"
                    Detail="Define a data-binding in code without a BindingContext"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:AlternativeCodeBindingPage}" />

                ...
            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>

```

When using commanding with XAML, `CommandParameter` properties are usually set to strings. In this case, however, a XAML markup extension is used so that the `CommandParameter` is of type `System.Type`.

Each `Command` property is bound to a property named `NavigateCommand`. That property is defined in the code-behind file, **MainPage.xaml.cs**:

```

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(
            async (Type pageType) =>
            {
                Page page = (Page)Activator.CreateInstance(pageType);
                await Navigation.PushAsync(page);
            });
    }

    BindingContext = this;
}

public ICommand NavigateCommand { private set; get; }
}

```

The constructor sets the `NavigateCommand` property to an `execute` method that instantiates the `System.Type` parameter and then navigates to it. Because the `PushAsync` call requires an `await` operator, the `execute` method must be flagged as asynchronous. This is accomplished with the `async` keyword before the parameter list.

The constructor also sets the `BindingContext` of the page to itself so that the bindings reference the

`NavigateCommand` in this class.

The order of the code in this constructor makes a difference: The `InitializeComponent` call causes the XAML to be parsed, but at that time the binding to a property named `NavigateCommand` cannot be resolved because `BindingContext` is set to `null`. If the `BindingContext` is set in the constructor *before* `NavigateCommand` is set, then the binding can be resolved when `BindingContext` is set, but at that time, `NavigateCommand` is still `null`. Setting `NavigateCommand` after `BindingContext` will have no effect on the binding because a change to `NavigateCommand` doesn't fire a `PropertyChanged` event, and the binding doesn't know that `NavigateCommand` is now valid.

Setting both `NavigateCommand` and `BindingContext` (in any order) prior to the call to `InitializeComponent` will work because both components of the binding are set when the XAML parser encounters the binding definition.

Data bindings can sometimes be tricky, but as you've seen in this series of articles, they are powerful and versatile, and help greatly to organize your code by separating underlying logic from the user interface.

## Related Links

- [Data Binding Demos \(sample\)](#)
- [Data binding chapter from Xamarin.Forms book](#)

# Xamarin.Forms Compiled Bindings

10/25/2018 • 6 minutes to read • [Edit Online](#)

*Compiled bindings are resolved more quickly than classic bindings, therefore improving data binding performance in Xamarin.Forms applications.*

Data bindings have two main problems:

1. There's no compile-time validation of binding expressions. Instead, bindings are resolved at runtime. Therefore, any invalid bindings aren't detected until runtime when the application doesn't behave as expected or error messages appear.
2. They aren't cost efficient. Bindings are resolved at runtime using general-purpose object inspection (reflection), and the overhead of doing this varies from platform to platform.

Compiled bindings improve data binding performance in Xamarin.Forms applications by resolving binding expressions at compile-time rather than runtime. In addition, this compile-time validation of binding expressions enables a better developer troubleshooting experience because invalid bindings are reported as build errors.

The process for using compiled bindings is to:

1. Enable XAML compilation. For more information about XAML compilation, see [XAML Compilation](#).
2. Set an `x:DataType` attribute on a `VisualElement` to the type of the object that the `VisualElement` and its children will bind to. Note that this attribute can be re-defined at any location in a view hierarchy.

## NOTE

It's recommended to set the `x:DataType` attribute at the same level in the view hierarchy as the `BindingContext` is set.

At XAML compile time, any invalid binding expressions will be reported as build errors. However, the XAML compiler will only report a build error for the first invalid binding expression that it encounters. Any valid binding expressions that are defined on the `VisualElement` or its children will be compiled, regardless of whether the `BindingContext` is set in XAML or code. Compiling a binding expression generates compiled code that will get a value from a property on the *source*, and set it on the property on the *target* that's specified in the markup. In addition, depending on the binding expression, the generated code may observe changes in the value of the *source* property and refresh the *target* property, and may push changes from the *target* back to the *source*.

## IMPORTANT

Compiled bindings are currently disabled for any binding expressions that define the `Source` property. This is because the `Source` property is always set using the `x:Reference` markup extension, which can't be resolved at compile time.

## Using compiled bindings

The [Compiled Color Selector](#) page demonstrates using compiled bindings between Xamarin.Forms views and ViewModel properties:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.CompiledColorSelectorPage"
    Title="Compiled Color Selector">
    ...
    <StackLayout x:DataType="local:HslColorViewModel">
        <StackLayout.BindingContext>
            <local:HslColorViewModel Color="Sienna" />
        </StackLayout.BindingContext>
        <BoxView Color="{Binding Color}"
            ... />
        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />
            <Slider Value="{Binding Hue}" />
            <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
            <Slider Value="{Binding Saturation}" />
            <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
            <Slider Value="{Binding Luminosity}" />
            <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>

```

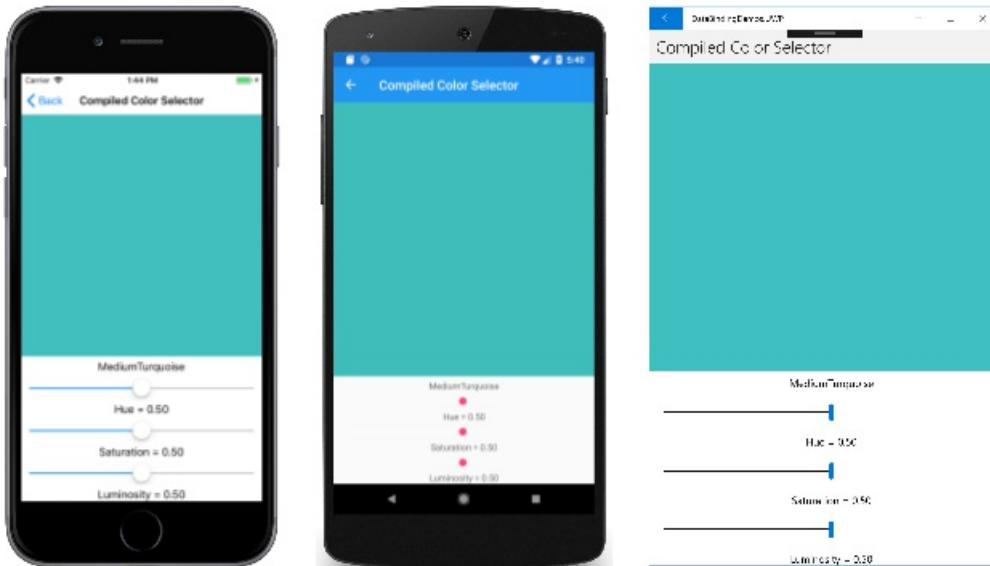
The root `StackLayout` instantiates the `HslColorViewModel` and initializes the `color` property within property element tags for the `BindingContext` property. This root `StackLayout` also defines the `x:DataType` attribute as the ViewModel type, indicating that any binding expressions in the root `StackLayout` view hierarchy will be compiled. This can be verified by changing any of the binding expressions to bind to a non-existent ViewModel property, which will result in a build error.

### IMPORTANT

The `x:DataType` attribute can be re-defined at any point in a view hierarchy.

The `BoxView`, `Label` elements, and `Slider` views inherit the binding context from the `StackLayout`. These views are all binding targets that reference source properties in the ViewModel. For the `BoxView.Color` property, and the `Label.Text` property, the data bindings are `OneWay` – the properties in the view are set from the properties in the ViewModel. However, the `Slider.Value` property uses a `TwoWay` binding. This allows each `Slider` to be set from the ViewModel, and also for the ViewModel to be set from each `Slider`.

When the application is first run, the `BoxView`, `Label` elements, and `Slider` elements are all set from the ViewModel based on the initial `Color` property set when the ViewModel was instantiated. This is shown in the following screenshots:



As the sliders are manipulated, the `BoxView` and `Label` elements are updated accordingly.

For more information about this color selector, see [ViewModels and Property-Change Notifications](#).

## Using compiled bindings in a DataTemplate

Bindings in a `DataTemplate` are interpreted in the context of the object being templated. Therefore, when using compiled bindings in a `DataTemplate`, the `DataTemplate` needs to declare the type of its data object using the `x:DataType` attribute.

The **Compiled Color List** page demonstrates using compiled bindings in a `DataTemplate`:

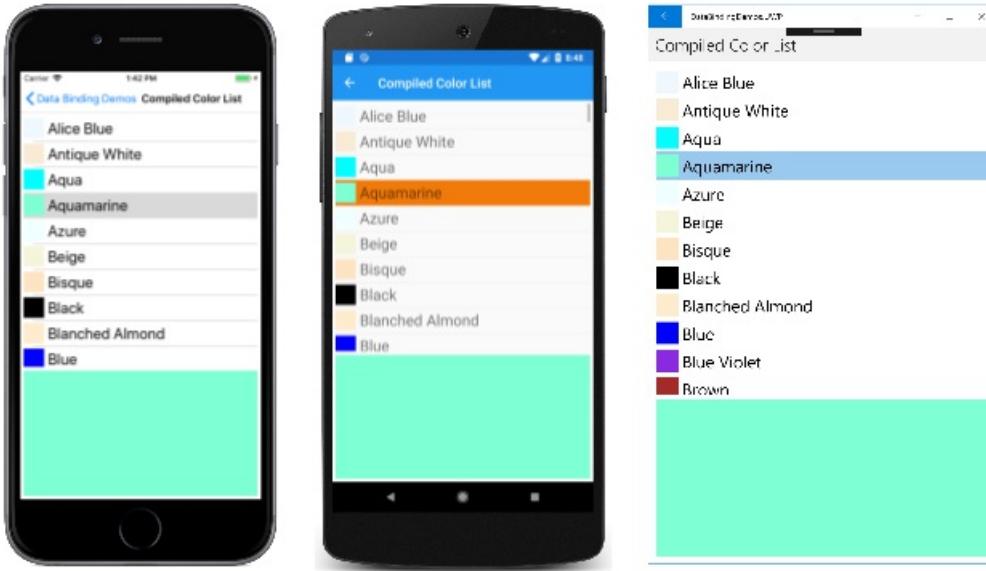
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.CompiledColorListPage"
    Title="Compiled Color List">
    <Grid>
        ...
        <ListView x:Name="colorListView"
            ItemsSource="{x:Static local:NamedColor.All}"
            ... >
            <ListView.ItemTemplate>
                <DataTemplate x:DataType="local:NamedColor">
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <BoxView Color="{Binding Color}">
                                ...
                            </BoxView>
                            <Label Text="{Binding FriendlyName}">
                                ...
                            </Label>
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
        <!-- The BoxView doesn't use compiled bindings -->
        <BoxView Color="{Binding Source={x:Reference colorListView}, Path=SelectedItem.Color}">
            ...
        </BoxView>
    </Grid>
</ContentPage>
```

The `ListView.ItemsSource` property is set to the static `NamedColor.All` property. The `NamedColor` class uses .NET reflection to enumerate all the static public fields in the `Color` structure, and to store them with their names in a

collection that is accessible from the static `All` property. Therefore, the `ListView` is filled with all of the `NamedColor` instances. For each item in the `ListView`, the binding context for the item is set to a `NamedColor` object. The `BoxView` and `Label` elements in the `ViewCell` are bound to `NamedColor` properties.

Note that the `DataTemplate` defines the `x:DataType` attribute to be the `NamedColor` type, indicating that any binding expressions in the `DataTemplate` view hierarchy will be compiled. This can be verified by changing any of the binding expressions to bind to a non-existent `NamedColor` property, which will result in a build error.

When the application is first run, the `ListView` is populated with `NamedColor` instances. When an item in the `ListView` is selected, the `BoxView.Color` property is set to the color of the selected item in the `ListView`:



Selecting other items in the `ListView` updates the color of the `BoxView`.

## Combining compiled bindings with classic bindings

Binding expressions are only compiled for the view hierarchy that the `x:DataType` attribute is defined on. Conversely, any views in a hierarchy on which the `x:DataType` attribute is not defined will use classic bindings. It's therefore possible to combine compiled bindings and classic bindings on a page. For example, in the previous section the views within the `DataTemplate` use compiled bindings, while the `BoxView` that's set to the color selected in the `ListView` does not.

Careful structuring of `x:DataType` attributes can therefore lead to a page using compiled and classic bindings. Alternatively, the `x:DataType` attribute can be re-defined at any point in a view hierarchy to `null` using the `x:Null` markup extension. Doing this indicates that any binding expressions within the view hierarchy will use classic bindings. The *Mixed Bindings* page demonstrates this approach:

```

<StackLayout x:DataType="local:HslColorViewModel">
    <StackLayout.BindingContext>
        <local:HslColorViewModel Color="Sienna" />
    </StackLayout.BindingContext>
    <BoxView Color="{Binding Color}"
        VerticalOptions="FillAndExpand" />
    <StackLayout x:DataType="{x:Null}"
        Margin="10, 0">
        <Label Text="{Binding Name}" />
        <Slider Value="{Binding Hue}" />
        <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
        <Slider Value="{Binding Saturation}" />
        <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
        <Slider Value="{Binding Luminosity}" />
        <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
    </StackLayout>
</StackLayout>

```

The root `StackLayout` sets the `x:DataType` attribute to be the `HslColorViewModel` type, indicating that any binding expression in the root `StackLayout` view hierarchy will be compiled. However, the inner `StackLayout` redefines the `x:DataType` attribute to `null` with the `x:Null` markup expression. Therefore, the binding expressions within the inner `StackLayout` use classic bindings. Only the `BoxView`, within the root `StackLayout` view hierarchy, uses compiled bindings.

For more information about the `x:Null` markup expression, see [x:Null Markup Extension](#).

## Performance

Compiled bindings improve data binding performance, with the performance benefit varying. Unit testing reveals that:

- A compiled binding that uses property-change notification (i.e. a `OneWay`, `OneWayToSource`, or `TwoWay` binding) is resolved approximately 8 times quicker than a classic binding.
- A compiled binding that doesn't use property-change notification (i.e. a `OneTime` binding) is resolved approximately 20 times quicker than a classic binding.
- Setting the `BindingContext` on a compiled binding that uses property change notification (i.e. a `OneWay`, `OneWayToSource`, or `TwoWay` binding) is approximately 5 times quicker than setting the `BindingContext` on a classic binding.
- Setting the `BindingContext` on a compiled binding that doesn't use property change notification (i.e. a `OneTime` binding) is approximately 7 times quicker than setting the `BindingContext` on a classic binding.

These performance differences can be magnified on mobile devices, dependent upon the platform being used, the version of the operating system being used, and the device on which the application is running.

## Related links

- [Data Binding Demos \(sample\)](#)

# Xamarin.Forms DependencyService

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms allows developers to define behavior in platform-specific projects. DependencyService then finds the right platform implementation, allowing shared code to access the native functionality.*

This guide is composed of the following articles:

- [\*\*Introduction\*\*](#) – introduces the overall architecture of the `DependencyService` concept.
- [\*\*Implementing Text-to-Speech\*\*](#) – walks through an example of using each platform's native text-to-speech system.
- [\*\*Checking Device Orientation\*\*](#) – walks through an example of using native platform APIs to determine the device's orientation.
- [\*\*Getting Battery Information\*\*](#) – walks through an example of using native APIs to get information on the battery's status.
- [\*\*Picking a Photo from the Library\*\*](#) – walks through an example of using native APIs to pick a photo from the phone's picture library.

## Related Links

- [Using DependencyService \(sample\)](#)
- [DependencyService \(sample\)](#)
- [Xamarin.Forms Samples](#)

# Introduction to DependencyService

10/31/2018 • 3 minutes to read • [Edit Online](#)

## Overview

`DependencyService` allows apps to call into platform-specific functionality from shared code. This functionality enables Xamarin.Forms apps to do anything that a native app can do.

`DependencyService` is a service locator. In practice, an interface is defined and `DependencyService` finds the correct implementation of that interface from the various platform projects.

### NOTE

By default, the `DependencyService` will only resolve platform implementations that have parameterless constructors. However, a dependency resolution method can be injected into Xamarin.Forms that uses a dependency injection container or factory methods to resolve platform implementations. This approach can be used to resolve platform implementations that have constructors with parameters. For more information, see [Dependency resolution in Xamarin.Forms](#).

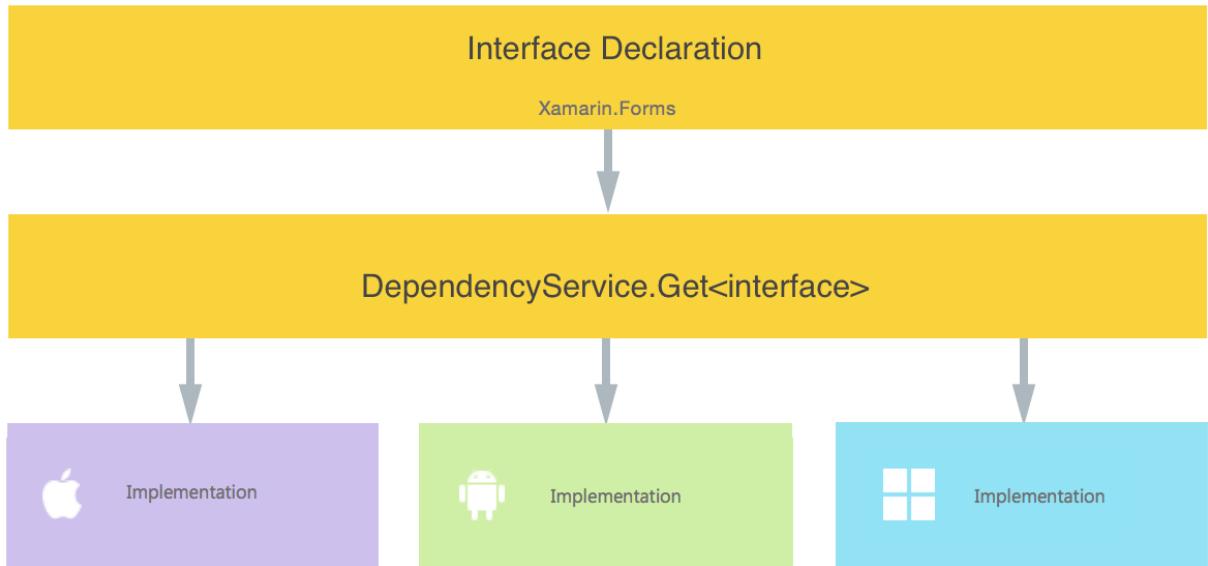
## How DependencyService Works

Xamarin.Forms apps need four components to use `DependencyService`:

- **Interface** – The required functionality is defined by an interface in shared code.
- **Implementation Per Platform** – Classes that implement the interface must be added to each platform project.
- **Registration** – Each implementing class must be registered with `DependencyService` via a metadata attribute. Registration enables `DependencyService` to find the implementing class and supply it in place of the interface at run time.
- **Call to DependencyService** – Shared code needs to explicitly call `DependencyService` to ask for implementations of the interface.

Note that implementations must be provided for each platform project in your solution. Platform projects without implementations will fail at runtime.

The structure of the application is explained by the following diagram:



## Interface

The interface you design will define how you interact with platform-specific functionality. Be careful if you are developing a component to be shared as a component or NuGet package. API design can make or break a package. The example below specifies a simple interface for speaking text that allows for flexibility in specifying the words to be spoken but leaves the implementation to be customized for each platform:

```

public interface ITextToSpeech {
    void Speak ( string text ); //note that interface members are public by default
}

```

## Implementation per Platform

Once a suitable interface has been designed, that interface must be implemented in the project for each platform that you are targeting. For example, the following class implements the `ITextToSpeech` interface on iOS:

```

namespace UsingDependencyService.iOS
{
    public class TextToSpeech_iOS : ITextToSpeech
    {
        public void Speak (string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer ();

            var speechUtterance = new AVSpeechUtterance (text) {
                Rate = AVSpeechUtterance.MaximumSpeechRate/4,
                Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance (speechUtterance);
        }
    }
}

```

## Registration

Each implementation of the interface needs to be registered with `DependencyService` with a metadata attribute. The following code registers the implementation for iOS:

```
[assembly: Dependency (typeof (TextToSpeech_iOS))]
namespace UsingDependencyService.iOS
{
    ...
}
```

Putting it all together, the platform-specific implementation looks like this:

```
[assembly: Dependency (typeof (TextToSpeech_iOS))]
namespace UsingDependencyService.iOS
{
    public class TextToSpeech_iOS : ITextToSpeech
    {
        public void Speak (string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer ();

            var speechUtterance = new AVSpeechUtterance (text) {
                Rate = AVSpeechUtterance.MaximumSpeechRate/4,
                Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance (speechUtterance);
        }
    }
}
```

Note: that the registration is performed at the namespace level, not the class level.

#### Universal Windows Platform .NET Native Compilation

UWP projects that use the .NET Native compilation option should follow a [slightly different configuration](#) when initializing Xamarin.Forms. .NET Native compilation also requires slightly different registration for dependency services.

In the **App.xaml.cs** file, manually register each dependency service defined in the UWP project using the `Register<T>` method, as shown below:

```
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
// register the dependencies in the same
Xamarin.Forms.DependencyService.Register<TextToSpeechImplementation>();
```

Note: manual registration using `Register<T>` is only effective in Release builds using .NET Native compilation. If you omit this line, Debug builds will still work, but Release builds will fail to load the dependency service.

#### Call to DependencyService

Once the project has been set up with a common interface and implementations for each platform, use `DependencyService` to get the right implementation at runtime:

```
DependencyService.Get<ITextToSpeech>().Speak("Hello from Xamarin Forms");
```

`DependencyService.Get<T>` will find the correct implementation of interface `T`.

#### Solution Structure

The [sample UsingDependencyService solution](#) is shown below for iOS and Android, with the code changes outlined above highlighted.

```

public interface ITextToSpeech
{
    void Speak (string text);
}

public class MainPage : ContentPage
{
    public MainPage ()
    {
        var speak = new Button {
            Text = "Hello, Forms !",
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.CenterAndExpand,
        };
        speak.Clicked += (sender, e) => {
            DependencyService.Get<ITextToSpeech>().Speak("Hello from Xamarin Forms");
        };
        Content = speak;
    }
}

[assembly: Dependency (typeof (TextToSpeech_Android))]

namespace UsingDependencyService.Android
{
    public class TextToSpeech_Android : Java.Lang.Object, ITextToSpeech, TextToSpeech
    {
        TextToSpeech speaker; string toSpeak;
        public TextToSpeech_Android () {}

        public void Speak (string text)
        {
            var c = Forms.Context;
            toSpeak = text;
            if (speaker == null) {
                speaker = new TextToSpeech (c, this);
            } else {
                var p = new Dictionary<string, string> ();
            }
        }
    }
}

[assembly: Dependency (typeof (TextToSpeech_iOS))]

namespace UsingDependencyService.iOS
{
    public class TextToSpeech_iOS : ITextToSpeech
    {
        public TextToSpeech_iOS () {}

        public void Speak (string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer ();

            var speechUtterance = new AVSpeechUtterance (text);
            Rate = AVSpeechUtterance.MaximumSpeechRate / 4;
            Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US");
        }
    }
}

```

#### NOTE

You **must** provide an implementation in every platform project. If no Interface implementation is registered, then the `DependencyService` will be unable to resolve the `Get<T>()` method at runtime.

## Related Links

- [DependencyServiceSample](#)
- [Xamarin.Forms Samples](#)

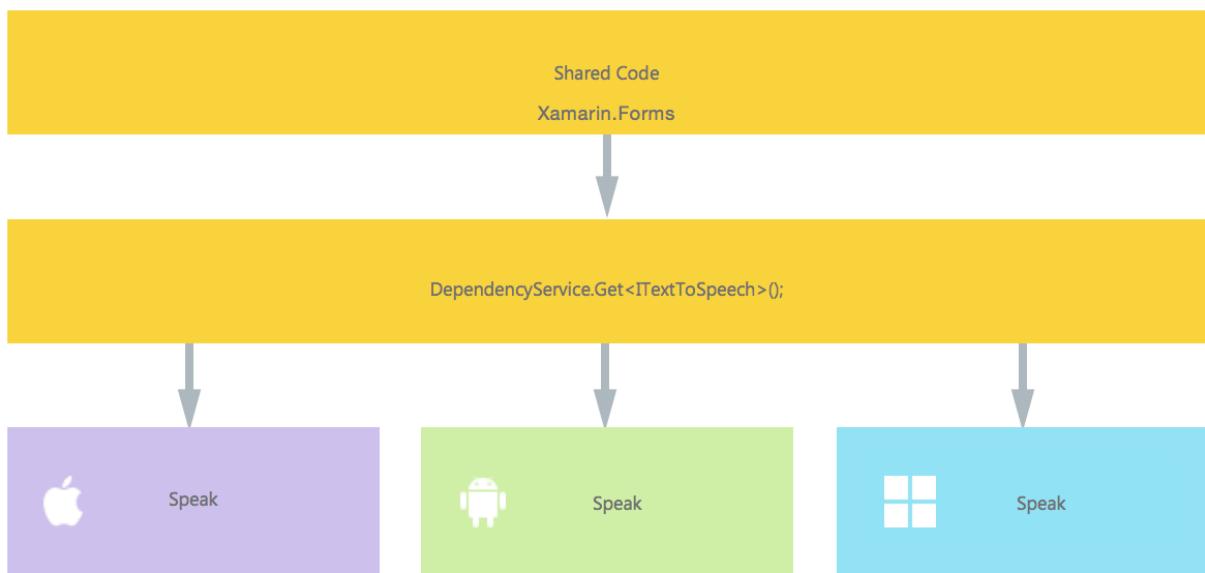
# Implementing Text-to-Speech

10/10/2018 • 3 minutes to read • [Edit Online](#)

This article will guide you as you create a cross-platform app that uses `DependencyService` to access native text-to-speech APIs:

- **Creating the Interface** – understand how the interface is created in shared code.
- **iOS Implementation** – learn how to implement the interface in native code for iOS.
- **Android Implementation** – learn how to implement the interface in native code for Android.
- **UWP Implementation** – learn how to implement the interface in native code for the Universal Windows Platform (UWP).
- **Implementing in Shared Code** – learn how to use `DependencyService` to call into the native implementation from shared code.

The application using `DependencyService` will have the following structure:



## Creating the Interface

First, create an interface in the shared code that expresses the functionality you plan to implement. For this example, the interface contains a single method, `Speak`:

```
public interface ITextToSpeech
{
    void Speak (string text);
}
```

Coding against this interface in the shared code will allow the `Xamarin.Forms` app to access the speech APIs on each platform.

### NOTE

Classes implementing the interface must have a parameterless constructor to work with the `DependencyService`.

## iOS Implementation

The interface must be implemented in each platform-specific application project. Note that the class has a parameterless constructor so that the `DependencyService` can create new instances.

```
[assembly: Dependency(typeof(TextToSpeechImplementation))]
namespace DependencyServiceSample.iOS
{

    public class TextToSpeechImplementation : ITextToSpeech
    {
        public TextToSpeechImplementation() { }

        public void Speak(string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer();
            var speechUtterance = new AVSpeechUtterance(text)
            {
                Rate = AVSpeechUtterance.MaximumSpeechRate / 4,
                Voice = AVSpeechSynthesisVoice.FromLanguage("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance(speechUtterance);
        }
    }
}
```

The `[assembly]` attribute registers the class as an implementation of the `ITextToSpeech` interface, which means that `DependencyService.Get<ITextToSpeech>()` can be used in the shared code to create an instance of it.

## Android Implementation

The Android code is more complex than the iOS version: it requires the implementing class to inherit from Android-specific `Java.Lang.Object` and to implement the `IOnInitListener` interface as well. It also requires access to the current Android context, which is exposed by the `MainActivity.Instance` property.

```
[assembly: Dependency(typeof(TextToSpeechImplementation))]
namespace DependencyServiceSample.Droid
{
    public class TextToSpeechImplementation : Java.Lang.Object, ITextToSpeech, TextToSpeech.IOnInitListener
    {
        TextToSpeech speaker;
        string toSpeak;

        public void Speak(string text)
        {
            toSpeak = text;
            if (speaker == null)
            {
                speaker = new TextToSpeech(MainActivity.Instance, this);
            }
            else
            {
                speaker.Speak(toSpeak, QueueMode.Flush, null, null);
            }
        }

        public void OnInit(OperationResult status)
        {
            if (status.Equals(OperationResult.Success))
            {
                speaker.Speak(toSpeak, QueueMode.Flush, null, null);
            }
        }
    }
}
```

The `[assembly]` attribute registers the class as an implementation of the `ITextToSpeech` interface, which means that `DependencyService.Get<ITextToSpeech>()` can be used in the shared code to create an instance of it.

## Universal Windows Platform Implementation

The Universal Windows Platform has a speech API in the `Windows.Media.SpeechSynthesis` namespace. The only caveat is to remember to tick the **Microphone** capability in the manifest, otherwise access to the speech APIs are blocked.

```
[assembly: Dependency(typeof(TextToSpeechImplementation))]
public class TextToSpeechImplementation : ITextToSpeech
{
    public async void Speak(string text)
    {
        var mediaElement = new MediaElement();
        var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
        var stream = await synth.SynthesizeTextToStreamAsync(text);

        mediaElement.SetSource(stream, stream.ContentType);
        mediaElement.Play();
    }
}
```

The `[assembly]` attribute registers the class as an implementation of the `ITextToSpeech` interface, which means that `DependencyService.Get<ITextToSpeech>()` can be used in the shared code to create an instance of it.

## Implementing in Shared Code

Now we can write and test shared code that accesses the text-to-speech interface. This simple page includes a button that triggers the speech functionality. It uses the `DependencyService` to get an instance of the `ITextToSpeech`

interface – at runtime this instance will be the platform-specific implementation that has full access to the native SDK.

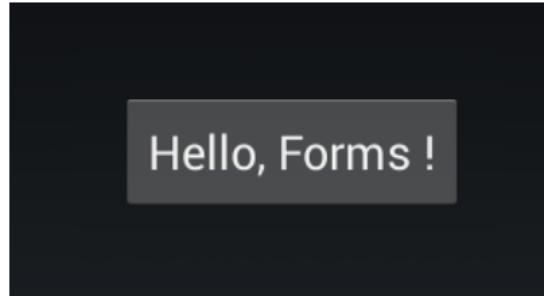
```
public MainPage ()  
{  
    var speak = new Button {  
        Text = "Hello, Forms !",  
        VerticalOptions = LayoutOptions.CenterAndExpand,  
        HorizontalOptions = LayoutOptions.CenterAndExpand,  
    };  
    speak.Clicked += (sender, e) => {  
        DependencyService.Get<ITextToSpeech>().Speak("Hello from Xamarin Forms");  
    };  
    Content = speak;  
}
```

Running this application on iOS, Android, or the UWP and pressing the button will result in the application speaking to you, using the native speech SDK on each platform.

iOS

Hello, Forms !

Android



## Related Links

- [Using DependencyService \(sample\)](#)
- [DependencyServiceSample](#)

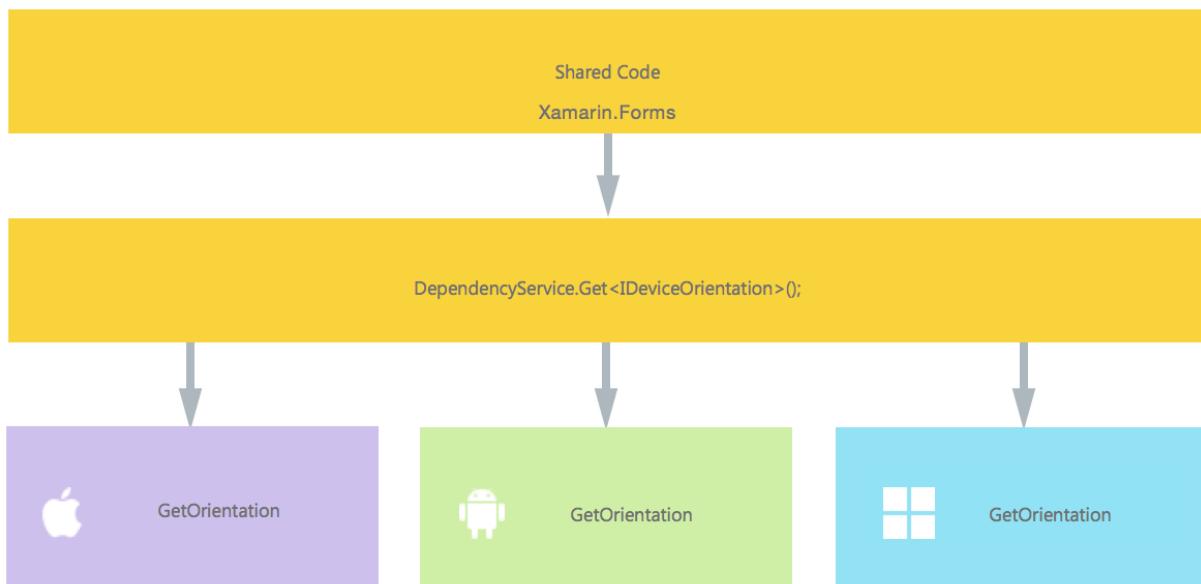
# Checking Device Orientation

10/9/2018 • 3 minutes to read • [Edit Online](#)

This article will guide you to use `DependencyService` to check the device orientation from shared code using the native APIs on each platform. This walkthrough is based on the existing `DeviceOrientation` plugin by Ali Özgür. See the [GitHub repo](#) for more information.

- **Creating the Interface** – understand how to the interface is created in shared code.
- **iOS Implementation** – learn how to implement the interface in native code for iOS.
- **Android Implementation** – learn how to implement the interface in native code for Android.
- **UWP Implementation** – learn how to implement the interface in native code for the Universal Windows Platform (UWP).
- **Implementing in Shared Code** – learn how to use `DependencyService` to call into the native implementation from shared code.

The application using `DependencyService` will have the following structure:



## NOTE

It is possible to detect whether the device is in portrait or landscape orientation in shared code, as demonstrated in [Device Orientation](#). The method described in this article uses native features to get more information about orientation, including whether the device is upside down.

## Creating the Interface

First, create an interface in the shared code that expresses the functionality you plan to implement. For this example, the interface contains a single method:

```

namespace DependencyServiceSample.Abstractions
{
    public enum DeviceOrientations
    {
        Undefined,
        Landscape,
        Portrait
    }

    public interface IDeviceOrientation
    {
        DeviceOrientations GetOrientation();
    }
}

```

Coding against this interface in the shared code will allow the Xamarin.Forms app to access the device orientation APIs on each platform.

#### NOTE

Classes implementing the interface must have a parameterless constructor to work with the `DependencyService`.

## iOS Implementation

The Interface must be implemented in each platform-specific application project. Note that the class has a parameterless constructor so that the `DependencyService` can create new instances:

```

using UIKit;
using Foundation;

namespace DependencyServiceSample.iOS
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation(){ }

        public DeviceOrientations GetOrientation()
        {
            var currentOrientation = UIApplication.SharedApplication.StatusBarOrientation;
            bool isPortrait = currentOrientation == UIInterfaceOrientation.Portrait
                || currentOrientation == UIInterfaceOrientation.PortraitUpsideDown;

            return isPortrait ? DeviceOrientations.Portrait: DeviceOrientations.Landscape;
        }
    }
}

```

Finally, add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements:

```

using UIKit;
using Foundation;
using DependencyServiceSample.iOS; //enables registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (DeviceOrientationImplementation))]
namespace DependencyServiceSample.iOS {
    ...
}

```

This attribute registers the class as an implementation of the `IDeviceOrientation` Interface, which means that

`DependencyService.Get<IDeviceOrientation>` can be used in the shared code to create an instance of it.

## Android Implementation

The following code implements `IDeviceOrientation` on Android:

```
using DependencyServiceSample.Droid;
using Android.Hardware;

namespace DependencyServiceSample.Droid
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation() { }

        public static void Init() { }

        public DeviceOrientations GetOrientation()
        {
            IWindowManager windowManager =
                Android.App.Application.Context.GetSystemService(Context.WindowService).JavaCast<IWindowManager>();

            var rotation = windowManager.DefaultDisplay.Rotation;
            bool isLandscape = rotation == SurfaceOrientation.Rotation90 || rotation ==
                SurfaceOrientation.Rotation270;
            return isLandscape ? DeviceOrientations.Landscape : DeviceOrientations.Portrait;
        }
    }
}
```

Add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements:

```
using DependencyServiceSample.Droid; //enables registration outside of namespace
using Android.Hardware;

[assembly: Xamarin.Forms.Dependency (typeof (DeviceOrientationImplementation))]
namespace DependencyServiceSample.Droid {
    ...
}
```

This attribute registers the class as an implementation of the `IDeviceOrientation` Interface, which means that `DependencyService.Get<IDeviceOrientation>` can be used in the shared code to create an instance of it.

## Universal Windows Platform Implementation

The following code implements the `IDeviceOrientation` interface on the Universal Windows Platform:

```

namespace DependencyServiceSample.WindowsPhone
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation() { }

        public DeviceOrientations GetOrientation()
        {
            var orientation = Windows.UI.ViewManagement.ApplicationView.GetForCurrentView().Orientation;
            if (orientation == Windows.UI.ViewManagement.ApplicationViewOrientation.Landscape) {
                return DeviceOrientations.Landscape;
            }
            else {
                return DeviceOrientations.Poratrait;
            }
        }
    }
}

```

Add the `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements:

```

using DependencyServiceSample.WindowsPhone; //enables registration outside of namespace

[assembly: Dependency(typeof(DeviceOrientationImplementation))]
namespace DependencyServiceSample.WindowsPhone {
    ...
}

```

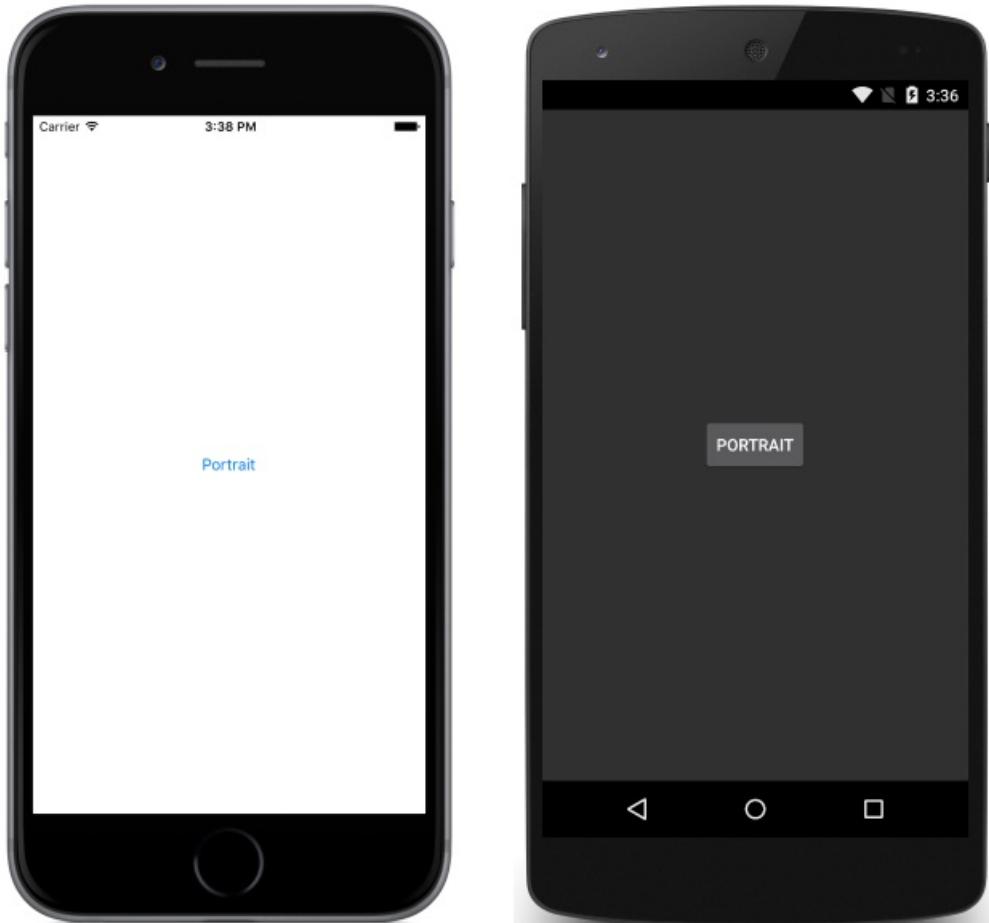
This attribute registers the class as an implementation of the `DeviceOrientationImplementation` Interface, which means that `DependencyService.Get<IDeviceOrientation>` can be used in the shared code to create an instance of it.

## Implementing in Shared Code

Now we can write and test shared code that accesses the `IDeviceOrientation` interface. This simple page includes a button that updates its own text based on the device orientation. It uses the `DependencyService` to get an instance of the `IDeviceOrientation` interface – at runtime this instance will be the platform-specific implementation that has full access to the native SDK:

```
public MainPage ()
{
    var orient = new Button {
        Text = "Get Orientation",
        VerticalOptions = LayoutOptions.CenterAndExpand,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
    orient.Clicked += (sender, e) => {
        var orientation = DependencyService.Get<IDeviceOrientation>().GetOrientation();
        switch(orientation){
            case DeviceOrientations.Undefined:
                orient.Text = "Undefined";
                break;
            case DeviceOrientations.Landscape:
                orient.Text = "Landscape";
                break;
            case DeviceOrientations.Portrait:
                orient.Text = "Portrait";
                break;
        }
    };
    Content = orient;
}
```

Running this application on iOS, Android, or the Windows platforms and pressing the button will result in the button's text updating with the device's orientation.



## Related Links

- [Using DependencyService \(sample\)](#)
- [DependencyService \(sample\)](#)

- [Xamarin.Forms Samples](#)

# Checking Battery Status

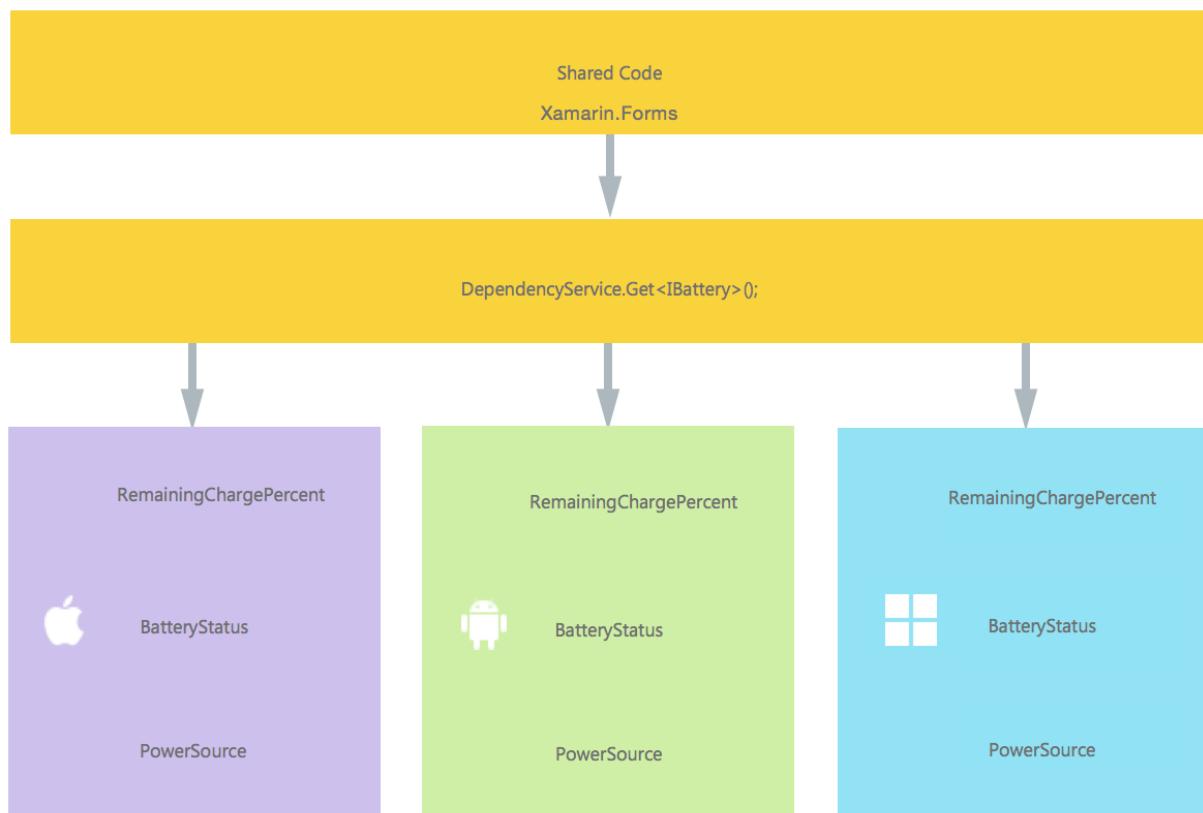
7/12/2018 • 6 minutes to read • [Edit Online](#)

This article walks through the creation of an application that checks battery status. This article is based on the Battery Plugin by James Montemagno. For more information, see the [GitHub repo](#).

Because Xamarin.Forms does not include functionality for checking the current battery status, this application will need to use `DependencyService` to take advantage of native APIs. This article will cover the following steps for using `DependencyService`:

- **Creating the Interface** – understand how the interface is created in shared code.
- **iOS Implementation** – learn how to implement the interface in native code for iOS.
- **Android Implementation** – learn how to implement the interface in native code for Android.
- **Universal Windows Platform Implementation** – learn how to implement the interface in native code for the Universal Windows Platform (UWP).
- **Implementing in Shared Code** – learn how to use `DependencyService` to call into the native implementation from shared code.

When completed, the application using `DependencyService` will have the following structure:



## Creating the Interface

First, create an interface in shared code that expresses the desired functionality. In the case of a battery checking application, the relevant information is the percentage of the battery remaining, whether the device is charging or not, and how the device is receiving power:

```
namespace DependencyServiceSample
{
    public enum BatteryStatus
    {
        Charging,
        Discharging,
        Full,
        NotCharging,
        Unknown
    }

    public enum PowerSource
    {
        Battery,
        Ac,
        Usb,
        Wireless,
        Other
    }

    public interface IBattery
    {
        int RemainingChargePercent { get; }
        BatteryStatus Status { get; }
        PowerSource PowerSource { get; }
    }
}
```

Coding against this interface in the shared code will allow the Xamarin.Forms app to access the power management APIs on each platform.

**NOTE**

Classes implementing the Interface must have a parameterless constructor to work with the `DependencyService`. Constructors can't be defined by interfaces.

## iOS Implementation

The `IBattery` interface must be implemented in each platform-specific application project. The iOS implementation will use the native `UIDevice` APIs to access battery information. Note that the following class has a parameterless constructor so that the `DependencyService` can create new instances:

```

using UIKit;
using Foundation;
using DependencyServiceSample.iOS;

namespace DependencyServiceSample.iOS
{
    public class BatteryImplementation : IBattery
    {
        public BatteryImplementation()
        {
            UIDevice.CurrentDevice.BatteryMonitoringEnabled = true;
        }

        public int RemainingChargePercent
        {
            get
            {
                return (int)(UIDevice.CurrentDevice.BatteryLevel * 100F);
            }
        }

        public BatteryStatus Status
        {
            get
            {
                switch (UIDevice.CurrentDevice.BatteryState)
                {
                    case UIDeviceBatteryState.Charging:
                        return BatteryStatus.Charging;
                    case UIDeviceBatteryState.Full:
                        return BatteryStatus.Full;
                    case UIDeviceBatteryState.Unplugged:
                        return BatteryStatus.Discharging;
                    default:
                        return BatteryStatus.Unknown;
                }
            }
        }

        public PowerSource PowerSource
        {
            get
            {
                switch (UIDevice.CurrentDevice.BatteryState)
                {
                    case UIDeviceBatteryState.Charging:
                        return PowerSource.Ac;
                    case UIDeviceBatteryState.Full:
                        return PowerSource.Ac;
                    case UIDeviceBatteryState.Unplugged:
                        return PowerSource.Battery;
                    default:
                        return PowerSource.Other;
                }
            }
        }
    }
}

```

Finally, add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements:

```

using UIKit;
using Foundation;
using DependencyServiceSample.iOS;//necessary for registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (BatteryImplementation))]
namespace DependencyServiceSample.iOS
{
    public class BatteryImplementation : IBattery {
        ...

```

This attribute registers the class as an implementation of the `IBattery` interface, which means that `DependencyService.Get<IBattery>` can be used in shared code to create an instance of it:

## Android Implementation

The Android implementation uses the [Android.OS.BatteryManager](#) API. This implementation is more complex than the iOS version, requiring checks to handle lack of battery permissions:

```

using System;
using Android;
using Android.Content;
using Android.App;
using Android.OS;
using BatteryStatus = Android.OS.BatteryStatus;
using DependencyServiceSample.Droid;

namespace DependencyServiceSample.Droid
{
    public class BatteryImplementation : IBattery
    {
        private BatteryBroadcastReceiver batteryReceiver;
        public BatteryImplementation() { }

        public int RemainingChargePercent
        {
            get
            {
                try
                {
                    using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
                    {
                        using (var battery = Application.Context.RegisterReceiver(null, filter))
                        {
                            var level = battery.GetIntExtra(BatteryManager.ExtraLevel, -1);
                            var scale = battery.GetIntExtra(BatteryManager.ExtraScale, -1);

                            return (int)Math.Floor(level * 100D / scale);
                        }
                    }
                }
                catch
                {
                    System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
                    throw;
                }
            }
        }

        public DependencyServiceSample.BatteryStatus Status
        {
            get
            {
                try

```

```

    {
        using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
        {
            using (var battery = Application.Context.RegisterReceiver(null, filter))
            {
                int status = battery.GetIntExtra(BatteryManager.ExtraStatus, -1);
                var isCharging = status == (int)BatteryStatus.Charging || status == (int)BatteryStatus.Full;

                var chargePlug = battery.GetIntExtra(BatteryManager.ExtraPlugged, -1);
                var usbCharge = chargePlug == (int)BatteryPlugged.Usb;
                var acCharge = chargePlug == (int)BatteryPlugged.Ac;
                bool wirelessCharge = false;
                wirelessCharge = chargePlug == (int)BatteryPlugged.Wireless;

                isCharging = (usbCharge || acCharge || wirelessCharge);
                if (isCharging)
                    return DependencyServiceSample.BatteryStatus.Charging;

                switch(status)
                {
                    case (int)BatteryStatus.Charging:
                        return DependencyServiceSample.BatteryStatus.Charging;
                    case (int)BatteryStatus.Discharging:
                        return DependencyServiceSample.BatteryStatus.Discharging;
                    case (int)BatteryStatus.Full:
                        return DependencyServiceSample.BatteryStatus.Full;
                    case (int)BatteryStatus.NotCharging:
                        return DependencyServiceSample.BatteryStatus.NotCharging;
                    default:
                        return DependencyServiceSample.BatteryStatus.Unknown;
                }
            }
        }
    }
    catch
    {
        System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
        throw;
    }
}

public PowerSource PowerSource
{
    get
    {
        try
        {
            using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
            {
                using (var battery = Application.Context.RegisterReceiver(null, filter))
                {
                    int status = battery.GetIntExtra(BatteryManager.ExtraStatus, -1);
                    var isCharging = status == (int)BatteryStatus.Charging || status == (int)BatteryStatus.Full;

                    var chargePlug = battery.GetIntExtra(BatteryManager.ExtraPlugged, -1);
                    var usbCharge = chargePlug == (int)BatteryPlugged.Usb;
                    var acCharge = chargePlug == (int)BatteryPlugged.Ac;

                    bool wirelessCharge = false;
                    wirelessCharge = chargePlug == (int)BatteryPlugged.Wireless;

                    isCharging = (usbCharge || acCharge || wirelessCharge);

                    if (!isCharging)
                        return DependencyServiceSample.PowerSource.Battery;
                    else if (usbCharge)
                        return DependencyServiceSample.PowerSource.Usb;
                    else if (acCharge)
                }
            }
        }
    }
}

```

```
        return DependencyServiceSample.PowerSource.Ac;
    else if (wirelessCharge)
        return DependencyServiceSample.PowerSource.Wireless;
    else
        return DependencyServiceSample.PowerSource.Other;
    }
}
}
catch
{
    System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
    throw;
}
}
}
}
}
```

Add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements:

```
...
using BatteryStatus = Android.OS.BatteryStatus;
using DependencyServiceSample.Droid; //enables registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (BatteryImplementation))]
namespace DependencyServiceSample.Droid
{
    public class BatteryImplementation : IBattery {
```

This attribute registers the class as an implementation of the `IBattery` interface, which means that `DependencyService.Get<IBattery>` can be used in the shared code to create an instance of it.

# Universal Windows Platform Implementation

The UWP implementation uses the `Windows.Devices.Power` APIs to obtain battery status information:

```
using DependencyServiceSample.UWP;
using Xamarin.Forms;

[assembly: Dependency(typeof(BatteryImplementation))]
namespace DependencyServiceSample.UWP
{
    public class BatteryImplementation : IBattery
    {
        private BatteryStatus status = BatteryStatus.Unknown;
        Windows.Devices.Power.Battery battery;

        public BatteryImplementation()
        {
        }

        private Windows.Devices.Power.Battery DefaultBattery
        {
            get
            {
                return battery ?? (battery = Windows.Devices.Power.Battery.AggregateBattery);
            }
        }

        public int RemainingChargePercent
        {

```

```

get
{
    var finalReport = DefaultBattery.GetReport();
    var finalPercent = -1;

    if (finalReport.RemainingCapacityInMilliwattHours.HasValue &&
finalReport.FullChargeCapacityInMilliwattHours.HasValue)
    {
        finalPercent = (int)((finalReport.RemainingCapacityInMilliwattHours.Value /
                           (double)finalReport.FullChargeCapacityInMilliwattHours.Value) * 100);
    }
    return finalPercent;
}

public BatteryStatus Status
{
    get
    {
        var report = DefaultBattery.GetReport();
        var percentage = RemainingChargePercent;

        if (percentage >= 1.0)
        {
            status = BatteryStatus.Full;
        }
        else if (percentage < 0)
        {
            status = BatteryStatus.Unknown;
        }
        else
        {
            switch (report.Status)
            {
                case Windows.System.Power.BatteryStatus.Charging:
                    status = BatteryStatus.Charging;
                    break;
                case Windows.System.Power.BatteryStatus.Discharging:
                    status = BatteryStatus.Discharging;
                    break;
                case Windows.System.Power.BatteryStatus.Idle:
                    status = BatteryStatus.NotCharging;
                    break;
                case Windows.System.Power.BatteryStatus.NotPresent:
                    status = BatteryStatus.Unknown;
                    break;
            }
        }
        return status;
    }
}

public PowerSource PowerSource
{
    get
    {
        if (status == BatteryStatus.Full || status == BatteryStatus.Charging)
        {
            return PowerSource.Ac;
        }
        return PowerSource.Battery;
    }
}
}

```

The `[assembly]` attribute above the namespace declaration registers the class as an implementation of the

`IBattery` interface, which means that `DependencyService.Get<IBattery>` can be used in shared code to create an instance of it.

## Implementing in Shared Code

Now that the interface has been implemented for each platform, the shared application can be written to take advantage of it. The application will consist of a page with a button that when tapped updates its text with the current battery status. It uses the `DependencyService` to get an instance of the `IBattery` interface. At runtime, this instance will be the platform-specific implementation that has full access to the native SDK.

```
public MainPage ()  
{  
    var button = new Button {  
        Text = "Click for battery info",  
        VerticalOptions = LayoutOptions.CenterAndExpand,  
        HorizontalOptions = LayoutOptions.CenterAndExpand,  
    };  
    button.Clicked += (sender, e) => {  
        var bat = DependencyService.Get<IBattery>();  
  
        switch (bat.PowerSource){  
            case PowerSource.Battery:  
                button.Text = "Battery - ";  
                break;  
            case PowerSource.Ac:  
                button.Text = "AC - ";  
                break;  
            case PowerSource.Usb:  
                button.Text = "USB - ";  
                break;  
            case PowerSource.Wireless:  
                button.Text = "Wireless - ";  
                break;  
            case PowerSource.Other:  
            default:  
                button.Text = "Other - ";  
                break;  
        }  
        switch (bat.Status){  
            case BatteryStatus.Charging:  
                button.Text += "Charging";  
                break;  
            case BatteryStatus.Discharging:  
                button.Text += "Discharging";  
                break;  
            case BatteryStatus.NotCharging:  
                button.Text += "Not Charging";  
                break;  
            case BatteryStatus.Full:  
                button.Text += "Full";  
                break;  
            case BatteryStatus.Unknown:  
            default:  
                button.Text += "Unknown";  
                break;  
        }  
    };  
    Content = button;  
}
```

Running this application on iOS, Android, or UWP and pressing the button will result in the button text updating to reflect the current power status of the device.



## Related Links

- [DependencyService \(sample\)](#)
- [Using DependencyService \(sample\)](#)
- [Xamarin.Forms Samples](#)

# Picking a Photo from the Picture Library

10/17/2018 • 6 minutes to read • [Edit Online](#)

This article walks through the creation of an application that allows the user to pick a photo from the phone's picture library. Because Xamarin.Forms does not include this functionality, it is necessary to use

`DependencyService` to access native APIs on each platform. This article will cover the following steps for using `DependencyService` for this task:

- **Creating the Interface** – understand how the interface is created in shared code.
- **iOS Implementation** – learn how to implement the interface in native code for iOS.
- **Android Implementation** – learn how to implement the interface in native code for Android.
- **Universal Windows Platform Implementation** – learn how to implement the interface in native code for the Universal Windows Platform (UWP).
- **Implementing in Shared Code** – learn how to use `DependencyService` to call into the native implementation from shared code.

## Creating the Interface

First, create an interface in shared code that expresses the desired functionality. In the case of a photo-picking application, just one method is required. This is defined in the `IPicturePicker` interface in the .NET Standard library of the sample code:

```
namespace DependencyServiceSample
{
    public interface IPicturePicker
    {
        Task<Stream> GetImageStreamAsync();
    }
}
```

The `GetImageStreamAsync` method is defined as asynchronous because the method must return quickly, but it can't return a `Stream` object for the selected photo until the user has browsed the picture library and selected one.

This interface is implemented in all the platforms using platform-specific code.

## iOS Implementation

The iOS implementation of the `IPicturePicker` interface uses the `UIImagePickerController` as described in the [Choose a Photo from the Gallery](#) recipe and [sample code](#).

The iOS implementation is contained in the `PicturePickerImplementation` class in the iOS project of the sample code. To make this class visible to the `DependencyService` manager, the class must be identified with an `[assembly]` attribute of type `Dependency`, and the class must be public and explicitly implement the `IPicturePicker` interface:

```

[assembly: Dependency (typeof (PicturePickerImplementation))]

namespace DependencyServiceSample.iOS
{
    public class PicturePickerImplementation : IPicturePicker
    {
        TaskCompletionSource<Stream> taskCompletionSource;
        UIImagePickerController imagePicker;

        public Task<Stream> GetImageStreamAsync()
        {
            // Create and define UIImagePickerController
            imagePicker = new UIImagePickerController
            {
                SourceType = UIImagePickerControllerSourceType.PhotoLibrary,
                MediaTypes =
                    UIImagePickerController.AvailableMediaTypes(UIImagePickerControllerSourceType.PhotoLibrary)
            };

            // Set event handlers
            imagePicker.FinishedPickingMedia += OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled += OnImagePickerCancelled;

            // Present UIImagePickerController;
            UIWindow window = UIApplication.SharedApplication.KeyWindow;
            var viewController = window.RootViewController;
            viewController.PresentModalViewController(imagePicker, true);

            // Return Task object
            taskCompletionSource = new TaskCompletionSource<Stream>();
            return taskCompletionSource.Task;
        }
        ...
    }
}

```

The `GetImageStreamAsync` method creates a `UIImagePickerController` and initializes it to select images from the photo library. Two event handlers are required: One for when the user selects a photo and the other for when the user cancels the display of the photo library. The `PresentModalViewController` then displays the photo library to the user.

At this point, the `GetImageStreamAsync` method must return a `Task<Stream>` object to the code that's calling it. This task is completed only when the user has finished interacting with the photo library and one of the event handlers is called. For situations like this, the `TaskCompletionSource` class is essential. The class provides a `Task` object of the proper generic type to return from the `GetImageStreamAsync` method, and the class can later be signaled when the task is completed.

The `FinishedPickingMedia` event handler is called when the user has selected a picture. However, the handler provides a `UIImage` object and the `Task` must return a .NET `Stream` object. This is done in two steps: The `UIImage` object is first converted to a JPEG file in memory stored in an `NSData` object, and then the `NSData` object is converted to a .NET `Stream` object. A call to the `SetResult` method of the `TaskCompletionSource` object completes the task by providing the `Stream` object:

```

namespace DependencyServiceSample.iOS
{
    public class PicturePickerImplementation : IPicturePicker
    {
        TaskCompletionSource<Stream> taskCompletionSource;
        UIImagePickerController imagePicker;
        ...
        void OnImagePickerFinishedPickingMedia(object sender, UIImagePickerMediaPickedEventArgs args)
        {
            UIImage image = args.EditedImage ?? args.OriginalImage;

            if (image != null)
            {
                // Convert UIImage to .NET Stream object
                NSData data = image.AsJPEG(1);
                Stream stream = dataAsStream();

                UnregisterEventHandlers();

                // Set the Stream as the completion of the Task
                taskCompletionSource.SetResult(stream);
            }
            else
            {
                UnregisterEventHandlers();
                taskCompletionSource.SetResult(null);
            }
            imagePicker.DismissModalViewController(true);
        }

        void OnImagePickerCancelled(object sender, EventArgs args)
        {
            UnregisterEventHandlers();
            taskCompletionSource.SetResult(null);
            imagePicker.DismissModalViewController(true);
        }

        void UnregisterEventHandlers()
        {
            imagePicker.FinishedPickingMedia -= OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled -= OnImagePickerCancelled;
        }
    }
}

```

An iOS application requires permission from the user to access the phone's photo library. Add the following to the `dict` section of the `Info.plist` file:

```

<key>NSPhotoLibraryUsageDescription</key>
<string>Picture Picker uses photo library</string>

```

## Android Implementation

The Android implementation uses the technique described in the [Select an Image](#) recipe and the [sample code](#). However, the method that is called when the user has selected an image from the picture library is an `OnActivityResult` override in a class that derives from `Activity`. For this reason, the normal `MainActivity` class in the Android project has been supplemented with a field, a property, and an override of the `OnActivityResult` method:

```

public class MainActivity : FormsAppCompatActivity
{
    ...
    // Field, property, and method for Picture Picker
    public static readonly int PickImageId = 1000;

    public TaskCompletionSource<Stream> PickImageTaskCompletionSource { set; get; }

    protected override void OnActivityResult(int requestCode, Result resultCode, Intent intent)
    {
        base.OnActivityResult(requestCode, resultCode, intent);

        if (requestCode == PickImageId)
        {
            if ((resultCode == Result.Ok) && (intent != null))
            {
                Android.Net.Uri uri = intent.Data;
                Stream stream = ContentResolver.OpenInputStream(uri);

                // Set the Stream as the completion of the Task
                PickImageTaskCompletionSource.SetResult(stream);
            }
            else
            {
                PickImageTaskCompletionSource.SetResult(null);
            }
        }
    }
}

```

The `OnActivityResult` override indicates the selected picture file with an Android `Uri` object, but this can be converted into a .NET `Stream` object by calling the `OpenInputStream` method of the `ContentResolver` object that was obtained from the activity's `ContentResolver` property.

Like the iOS implementation, the Android implementation uses a `TaskCompletionSource` to signal when the task has been completed. This `TaskCompletionSource` object is defined as a public property in the `MainActivity` class. This allows the property to be referenced in the `PicturePickerImplementation` class in the Android project. This is the class with the `GetImageStreamAsync` method:

```
[assembly: Dependency(typeof(PicturePickerImplementation))]

namespace DependencyServiceSample.Droid
{
    public class PicturePickerImplementation : IPicturePicker
    {
        public Task<Stream> GetImageStreamAsync()
        {
            // Define the Intent for getting images
            Intent intent = new Intent();
            intent.SetType("image/*");
            intent.SetAction(Intent.ActionGetContent);

            // Start the picture-picker activity (resumes in MainActivity.cs)
            MainActivity.Instance.StartActivityForResult(
                Intent.CreateChooser(intent, "Select Picture"),
                MainActivity.PickImageId);

            // Save the TaskCompletionSource object as a MainActivity property
            MainActivity.Instance.PickImageTaskCompletionSource = new TaskCompletionSource<Stream>();

            // Return Task object
            return MainActivity.Instance.PickImageTaskCompletionSource.Task;
        }
    }
}
```

This method accesses the `MainActivity` class for several purposes: for the `Instance` property, for the `PickImageId` field, for the `TaskCompletionSource` property, and to call `StartActivityForResult`. This method is defined by the `FormsAppCompatActivity` class, which is the base class of `MainActivity`.

## UWP Implementation

Unlike the iOS and Android implementations, the implementation of the photo picker for the Universal Windows Platform does not require the `TaskCompletionSource` class. The `PicturePickerImplementation` class uses the `FileOpenPicker` class to get access to the photo library. Because the `PickSingleFileAsync` method of `FileOpenPicker` is itself asynchronous, the `GetImageStreamAsync` method can simply use `await` with that method (and other asynchronous methods) and return a `Stream` object:

```
[assembly: Dependency(typeof(PicturePickerImplementation))]

namespace DependencyServiceSample.UWP
{
    public class PicturePickerImplementation : IPicturePicker
    {
        public async Task<Stream> GetImageStreamAsync()
        {
            // Create and initialize the FileOpenPicker
            FileOpenPicker openPicker = new FileOpenPicker
            {
                ViewMode = PickerViewMode.Thumbnail,
                SuggestedStartLocation = PickerLocationId.PicturesLibrary,
            };

            openPicker.FileTypeFilter.Add(".jpg");
            openPicker.FileTypeFilter.Add(".jpeg");
            openPicker.FileTypeFilter.Add(".png");

            // Get a file and return a Stream
            StorageFile storageFile = await openPicker.PickSingleFileAsync();

            if (storageFile == null)
            {
                return null;
            }

            IRandomAccessStreamWithContentType raStream = await storageFile.OpenReadAsync();
            return raStream.AsStreamForRead();
        }
    }
}
```

## Implementing in Shared Code

Now that the interface has been implemented for each platform, the application in the .NET Standard library can take advantage of it.

The `App` class creates a `Button` to pick a photo:

```
Button pickPictureButton = new Button
{
    Text = "Pick Photo",
    VerticalOptions = LayoutOptions.CenterAndExpand,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
stack.Children.Add(pickPictureButton);
```

The `clicked` handler uses the `DependencyService` class to call `GetImageStreamAsync`. This results in a call in the platform project. If the method returns a `Stream` object, then the handler creates an `Image` element for that picture with a `TapGestureRecognizer`, and replaces the `StackLayout` on the page with that `Image`:

```
pickPictureButton.Clicked += async (sender, e) =>
{
    pickPictureButton.IsEnabled = false;
    Stream stream = await DependencyService.Get<IPicturePicker>().GetImageStreamAsync();

    if (stream != null)
    {
        Image image = new Image
        {
            Source = ImageSource.FromStream(() => stream),
            BackgroundColor = Color.Gray
        };

        TapGestureRecognizer recognizer = new TapGestureRecognizer();
        recognizer.Tapped += (sender2, args) =>
        {
            (MainPage as ContentPage).Content = stack;
            pickPictureButton.IsEnabled = true;
        };
        image.GestureRecognizers.Add(recognizer);

        (MainPage as ContentPage).Content = image;
    }
    else
    {
        pickPictureButton.IsEnabled = true;
    }
};
```

Tapping the `Image` element returns the page to normal.

## Related Links

- [Choose a Photo from the Gallery \(iOS\)](#)
- [Select an Image \(Android\)](#)
- [DependencyService \(sample\)](#)

# Xamarin.Forms Effects

7/12/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms user interfaces are rendered using the native controls of the target platform, allowing Xamarin.Forms applications to retain the appropriate look and feel for each platform. Effects allow the native controls on each platform to be customized without having to resort to a custom renderer implementation.*

## Introduction to Effects

Effects allow the native controls on each platform to be customized, and are typically used for small styling changes. This article provides an introduction to effects, outlines the boundary between effects and custom renderers, and describes the `PlatformEffect` class.

## Creating an Effect

Effects simplify the customization of a control. This article demonstrates how to create an effect that changes the background color of the `Entry` control when the control gains focus.

## Passing Parameters to an Effect

Creating an effect that's configured through parameters enables the effect to be reused. These articles demonstrate using properties to pass parameters to an effect, and changing a parameter at runtime.

## Invoking Events from an Effect

Effects can invoke events. This article shows how to create an event that implements low-level multi-touch finger tracking and signals an application for touch presses, moves, and releases.

# Introduction to Effects

7/12/2018 • 3 minutes to read • [Edit Online](#)

Effects allow the native controls on each platform to be customized, and are typically used for small styling changes. This article provides an introduction to effects, outlines the boundary between effects and custom renderers, and describes the `PlatformEffect` class.

Xamarin.Forms [Pages, Layouts and Controls](#) presents a common API to describe cross-platform mobile user interfaces. Each page, layout, and control is rendered differently on each platform using a `Renderer` class that in turn creates a native control (corresponding to the Xamarin.Forms representation), arranges it on the screen, and adds the behavior specified in the shared code.

Developers can implement their own custom `Renderer` classes to customize the appearance and/or behavior of a control. However, implementing a custom renderer class to perform a simple control customization is often a heavy-weight response. Effects simplify this process, allowing the native controls on each platform to be more easily customized.

Effects are created in platform-specific projects by subclassing the `PlatformEffect` control, and then the effects are consumed by attaching them to an appropriate control in a Xamarin.Forms .NET Standard library or Shared Library project.

## Why Use an Effect over a Custom Renderer?

Effects simplify the customization of a control, are reusable, and can be parameterized to further increase reuse.

Anything that can be achieved with an effect can also be achieved with a custom renderer. However, custom renderers offer more flexibility and customization than effects. The following guidelines list the circumstances in which to choose an effect over a custom renderer:

- An effect is recommended when changing the properties of a platform-specific control will achieve the desired result.
- A custom renderer is required when there's a need to override methods of a platform-specific control.
- A custom renderer is required when there's a need to replace the platform-specific control that implements a Xamarin.Forms control.

## Subclassing the PlatformEffect Class

The following table lists the namespace for the `PlatformEffect` class on each platform, and the types of its properties:

PLATFORM	NAMESPACE	CONTAINER	CONTROL
iOS	Xamarin.Forms.Platform.iOS	UIView	UIView
Android	Xamarin.Forms.Platform.Android	ViewGroup	View
Universal Windows Platform (UWP)	Xamarin.Forms.Platform.UWP	FrameworkElement	FrameworkElement

Each platform-specific `PlatformEffect` class exposes the following properties:

- `Container` – references the platform-specific control being used to implement the layout.
- `Control` – references the platform-specific control being used to implement the Xamarin.Forms control.
- `Element` – references the Xamarin.Forms control that's being rendered.

Effects do not have type information about the container, control, or element they are attached to because they can be attached to any element. Therefore, when an effect is attached to an element that it doesn't support it should degrade gracefully or throw an exception. However, the `Container`, `Control`, and `Element` properties can be cast to their implementing type. For more information about these types see [Renderer Base Classes and Native Controls](#).

Each platform-specific `PlatformEffect` class exposes the following methods, which must be overridden to implement an effect:

- `OnAttached` – called when an effect is attached to a Xamarin.Forms control. An overridden version of this method, in each platform-specific effect class, is the place to perform customization of the control, along with exception handling in case the effect cannot be applied to the specified Xamarin.Forms control.
- `OnDetached` – called when an effect is detached from a Xamarin.Forms control. An overridden version of this method, in each platform-specific effect class, is the place to perform any effect cleanup such as de-registering an event handler.

In addition, the `PlatformEffect` exposes the `OnElementPropertyChanged` method, which can also be overridden. This method is called when a property of the element has changed. An overridden version of this method, in each platform-specific effect class, is the place to respond to bindable property changes on the Xamarin.Forms control. A check for the property that's changed should always be made, as this override can be called many times.

## Related Links

- [Custom Renderers](#)

# Creating an Effect

7/12/2018 • 6 minutes to read • [Edit Online](#)

Effects simplify the customization of a control. This article demonstrates how to create an effect that changes the background color of the `Entry` control when the control gains focus.

The process for creating an effect in each platform-specific project is as follows:

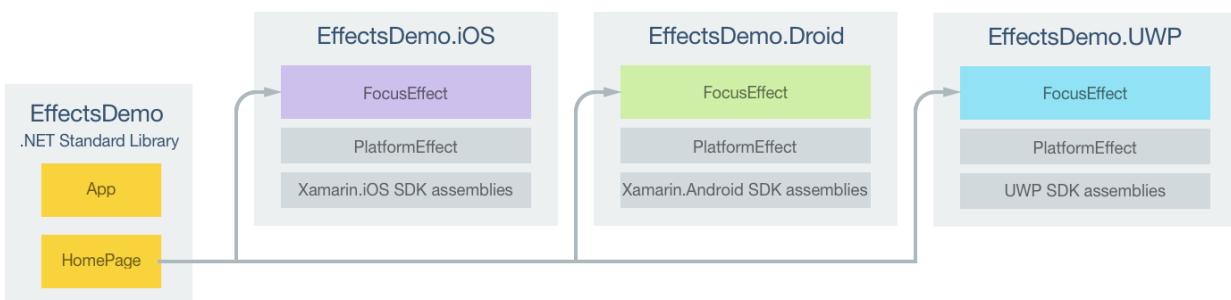
1. Create a subclass of the `PlatformEffect` class.
2. Override the `OnAttached` method and write logic to customize the control.
3. Override the `OnDetached` method and write logic to clean up the control customization, if required.
4. Add a `ResolutionGroupName` attribute to the effect class. This attribute sets a company wide namespace for effects, preventing collisions with other effects with the same name. Note that this attribute can only be applied once per project.
5. Add an `ExportEffect` attribute to the effect class. This attribute registers the effect with a unique ID that's used by Xamarin.Forms, along with the group name, to locate the effect prior to applying it to a control. The attribute takes two parameters – the type name of the effect, and a unique string that will be used to locate the effect prior to applying it to a control.

The effect can then be consumed by attaching it to the appropriate control.

## NOTE

It's optional to provide an effect in each platform project. Attempting to use an effect when one isn't registered will return a non-null value that does nothing.

The sample application demonstrates a `FocusEffect` that changes the background color of a control when it gains focus. The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



An `Entry` control on the `HomePage` is customized by the `FocusEffect` class in each platform-specific project. Each `FocusEffect` class derives from the `PlatformEffect` class for each platform. This results in the `Entry` control being rendered with a platform-specific background color, which changes when the control gains focus, as shown in the following screenshots:

Effect attached to an Entry

iOS

Effect attached to an Entry

Android

Effect attached to an Entry

iOS

Effect attached to an Entry

Android

## Creating the Effect on Each Platform

The following sections discuss the platform-specific implementation of the `FocusEffect` class.

### iOS Project

The following code example shows the `FocusEffect` implementation for the iOS project:

```

using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ResolutionGroupName ("MyCompany")]
[assembly: ExportEffect (typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.iOS
{
    public class FocusEffect : PlatformEffect
    {
        UIColor backgroundColor;

        protected override void OnAttached ()
        {
            try {
                Control.BackgroundColor = backgroundColor = UIColor.FromRGB (204, 153, 255);
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }

        protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged (args);

            try {
                if (args.PropertyName == "IsFocused") {
                    if (Control.BackgroundColor == backgroundColor) {
                        Control.BackgroundColor = UIColor.White;
                    } else {
                        Control.BackgroundColor = backgroundColor;
                    }
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}

```

The `OnAttached` method sets the `BackgroundColor` property of the control to light purple with the `UIColor.FromRGB` method, and also stores this color in a field. This functionality is wrapped in a `try / catch` block in case the control the effect is attached to does not have a `BackgroundColor` property. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

The `OnElementPropertyChanged` override responds to bindable property changes on the `Xamarin.Forms` control. When the `IsFocused` property changes, the `BackgroundColor` property of the control is changed to white if the control has focus, otherwise it's changed to light purple. This functionality is wrapped in a `try / catch` block in case the control the effect is attached to does not have a `BackgroundColor` property.

## Android Project

The following code example shows the `FocusEffect` implementation for the Android project:

```

using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ResolutionGroupName ("MyCompany")]
[assembly: ExportEffect (typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.Droid
{
    public class FocusEffect : PlatformEffect
    {
        Android.Graphics.Color backgroundColor;

        protected override void OnAttached ()
        {
            try {
                backgroundColor = Android.Graphics.Color.LightGreen;
                Control.SetBackgroundColor (backgroundColor);

            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }

        protected override void OnElementPropertyChanged (System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged (args);
            try {
                if (args.PropertyName == "IsFocused") {
                    if (((Android.Graphics.Drawables.ColorDrawable)Control.Background).Color ==
backgroundColor) {
                        Control.SetBackgroundColor (Android.Graphics.Color.Black);
                    } else {
                        Control.SetBackgroundColor (backgroundColor);
                    }
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}

```

The `OnAttached` method calls the `SetBackgroundColor` method to set the background color of the control to light green, and also stores this color in a field. This functionality is wrapped in a `try / catch` block in case the control the effect is attached to does not have a `SetBackgroundColor` property. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

The `OnElementPropertyChanged` override responds to bindable property changes on the `Xamarin.Forms` control. When the `IsFocused` property changes, the background color of the control is changed to white if the control has focus, otherwise it's changed to light green. This functionality is wrapped in a `try / catch` block in case the control the effect is attached to does not have a `BackgroundColor` property.

## Universal Windows Platform Projects

The following code example shows the `FocusEffect` implementation for Universal Windows Platform (UWP) projects:

```

using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.UWP
{
    public class FocusEffect : PlatformEffect
    {
        protected override void OnAttached()
        {
            try
            {
                (Control as Windows.UI.Xaml.Controls.Control).Background = new SolidColorBrush(Colors.Cyan);
                (Control as FormsTextBox).BackgroundFocusBrush = new SolidColorBrush(Colors.White);
            }
            catch (Exception ex)
            {
                Debug.WriteLine("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached()
        {
        }
    }
}

```

The `OnAttached` method sets the `Background` property of the control to cyan, and sets the `BackgroundFocusBrush` property to white. This functionality is wrapped in a `try / catch` block in case the control lacks these properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

## Consuming the Effect

The process for consuming an effect from a `Xamarin.Forms` .NET Standard library or Shared Library project is as follows:

1. Declare a control that will be customized by the effect.
2. Attach the effect to the control by adding it to the control's `Effects` collection.

### NOTE

An effect instance can only be attached to a single control. Therefore, an effect must be resolved twice to use it on two controls.

## Consuming the Effect in XAML

The following XAML code example shows an `Entry` control to which the `FocusEffect` is attached:

```

<Entry Text="Effect attached to an Entry" ...>
    <Entry.Effects>
        <local:FocusEffect />
    </Entry.Effects>
    ...
</Entry>

```

The `FocusEffect` class in the .NET Standard library supports effect consumption in XAML, and is shown in the

following code example:

```
public class FocusEffect : RoutingEffect
{
    public FocusEffect () : base ("MyCompany.FocusEffect")
    {
    }
}
```

The `FocusEffect` class subclasses the `RoutingEffect` class, which represents a platform-independent effect that wraps an inner effect that is usually platform-specific. The `FocusEffect` class calls the base class constructor, passing in a parameter consisting of a concatenation of the resolution group name (specified using the `ResolutionGroupName` attribute on the effect class), and the unique ID that was specified using the `ExportEffect` attribute on the effect class. Therefore, when the `Entry` is initialized at runtime, a new instance of the `MyCompany.FocusEffect` is added to the control's `Effects` collection.

Effects can also be attached to controls by using a behavior, or by using attached properties. For more information about attaching an effect to a control by using a behavior, see [Reusable EffectBehavior](#). For more information about attaching an effect to a control by using attached properties, see [Passing Parameters to an Effect](#).

## Consuming the Effect in C#

The equivalent `Entry` in C# is shown in the following code example:

```
var entry = new Entry {
    Text = "Effect attached to an Entry",
    ...
};
```

The `FocusEffect` is attached to the `Entry` instance by adding the effect to the control's `Effects` collection, as demonstrated in the following code example:

```
public HomePageCS ()
{
    ...
    entry.Effects.Add (Effect.Resolve ("MyCompany.FocusEffect"));
    ...
}
```

The `Effect.Resolve` returns an `Effect` for the specified name, which is a concatenation of the resolution group name (specified using the `ResolutionGroupName` attribute on the effect class), and the unique ID that was specified using the `ExportEffect` attribute on the effect class. If a platform doesn't provide the effect, the `Effect.Resolve` method will return a non-`null` value.

## Summary

This article demonstrated how to create an effect that changes the background color of the `Entry` control when the control gains focus.

## Related Links

- [Custom Renderers](#)
- [Effect](#)
- [PlatformEffect](#)

- [Background Color Effect \(sample\)](#)
- [Focus Effect \(sample\)](#)

# Passing Parameters to an Effect

4/12/2018 • 2 minutes to read • [Edit Online](#)

*Effect parameters can be defined by properties, enabling the effect to be reused. Parameters can then be passed to the effect by specifying values for each property when instantiating the effect.*

## Passing Effect Parameters as Common Language Runtime Properties

Common Language Runtime (CLR) properties can be used to define effect parameters that don't respond to runtime property changes. This article demonstrates using CLR properties to pass parameters to an effect.

## Passing Effect Parameters as Attached Properties

Attached properties can be used to define effect parameters that respond to runtime property changes. This article demonstrates using attached properties to pass parameters to an effect, and changing a parameter at runtime.

# Passing Effect Parameters as Common Language Runtime Properties

7/12/2018 • 5 minutes to read • [Edit Online](#)

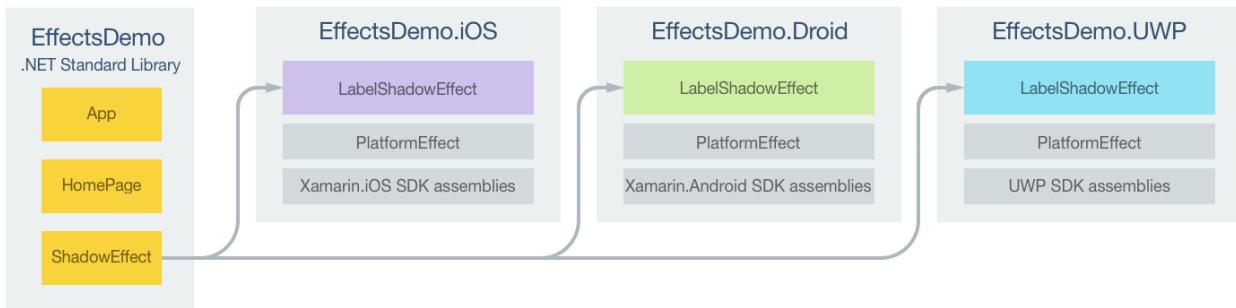
*Common Language Runtime (CLR) properties can be used to define effect parameters that don't respond to runtime property changes. This article demonstrates using CLR properties to pass parameters to an effect.*

The process for creating effect parameters that don't respond to runtime property changes is as follows:

1. Create a `public` class that subclasses the `RoutingEffect` class. The `RoutingEffect` class represents a platform-independent effect that wraps an inner effect that is usually platform-specific.
2. Create a constructor that calls the base class constructor, passing in a concatenation of the resolution group name, and the unique ID that was specified on each platform-specific effect class.
3. Add properties to the class for each parameter to be passed to the effect.

Parameters can then be passed to the effect by specifying values for each property when instantiating the effect.

The sample application demonstrates a `ShadowEffect` that adds a shadow to the text displayed by a `Label` control. The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



A `Label` control on the `HomePage` is customized by the `LabelShadowEffect` in each platform-specific project. Parameters are passed to each `LabelShadowEffect` through properties in the `ShadowEffect` class. Each `LabelShadowEffect` class derives from the `PlatformEffect` class for each platform. This results in a shadow being added to the text displayed by the `Label` control, as shown in the following screenshots:

Label Shadow Effect

iOS

Label Shadow Effect

Android

## Creating Effect Parameters

A `public` class that subclasses the `RoutingEffect` class should be created to represent effect parameters, as demonstrated in the following code example:

```

public class ShadowEffect : RoutingEffect
{
    public float Radius { get; set; }

    public Color Color { get; set; }

    public float DistanceX { get; set; }

    public float DistanceY { get; set; }

    public ShadowEffect () : base ("MyCompany.LabelShadowEffect")
    {
    }
}

```

The `ShadowEffect` contains four properties that represent parameters to be passed to each platform-specific `LabelShadowEffect`. The class constructor calls the base class constructor, passing in a parameter consisting of a concatenation of the resolution group name, and the unique ID that was specified on each platform-specific effect class. Therefore, a new instance of the `MyCompany.LabelShadowEffect` will be added to a control's `Effects` collection when a `ShadowEffect` is instantiated.

## Consuming the Effect

The following XAML code example shows a `Label` control to which the `ShadowEffect` is attached:

```

<Label Text="Label Shadow Effect" ...>
    <Label.Effects>
        <local:ShadowEffect Radius="5" DistanceX="5" DistanceY="5">
            <local:ShadowEffect.Color>
                <OnPlatform x:TypeArguments="Color">
                    <On Platform="iOS" Value="Black" />
                    <On Platform="Android" Value="White" />
                    <On Platform="UWP" Value="Red" />
                </OnPlatform>
            </local:ShadowEffect.Color>
        </local:ShadowEffect>
    </Label.Effects>
</Label>

```

The equivalent `Label` in C# is shown in the following code example:

```
var label = new Label {
    Text = "Label Shadow Effect",
    ...
};

Color color = Color.Default;
switch (Device.RuntimePlatform)
{
    case Device.iOS:
        color = Color.Black;
        break;
    case Device.Android:
        color = Color.White;
        break;
    case Device.UWP:
        color = Color.Red;
        break;
}

label.Effects.Add (new ShadowEffect {
    Radius = 5,
    Color = color,
    DistanceX = 5,
    DistanceY = 5
});
```

In both code examples, an instance of the `ShadowEffect` class is instantiated with values being specified for each property, before being added to the control's `Effects` collection. Note that the `ShadowEffect.Color` property uses platform-specific color values. For more information, see [Device Class](#).

## Creating the Effect on each Platform

The following sections discuss the platform-specific implementation of the `LabelShadowEffect` class.

### iOS Project

The following code example shows the `LabelShadowEffect` implementation for the iOS project:

```

[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                if (effect != null) {
                    Control.Layer.CornerRadius = effect.Radius;
                    Control.Layer.ShadowColor = effect.Color.ToCGColor ();
                    Control.Layer.ShadowOffset = new CGSize (effect.DistanceX, effect.DistanceY);
                    Control.Layer.ShadowOpacity = 1.0f;
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}

```

The `OnAttached` method retrieves the `ShadowEffect` instance, and sets `Control.Layer` properties to the specified property values to create the shadow. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

## Android Project

The following code example shows the `LabelShadowEffect` implementation for the Android project:

```

[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.Droid
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                var control = Control as Android.Widget.TextView;
                var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                if (effect != null) {
                    float radius = effect.Radius;
                    float distanceX = effect.DistanceX;
                    float distanceY = effect.DistanceY;
                    Android.Graphics.Color color = effect.Color.ToAndroid ();
                    control.SetShadowLayer (radius, distanceX, distanceY, color);
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}

```

The `OnAttached` method retrieves the `ShadowEffect` instance, and calls the `TextView.SetShadowLayer` method to create a shadow using the specified property values. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

## Universal Windows Platform Project

The following code example shows the `LabelShadowEffect` implementation for the Universal Windows Platform (UWP) project:

```
[assembly: ResolutionGroupName ("Xamarin")]
[assembly: ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.UWP
{
    public class LabelShadowEffect : PlatformEffect
    {
        bool shadowAdded = false;

        protected override void OnAttached ()
        {
            try {
                if (!shadowAdded) {
                    var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                    if (effect != null) {
                        var textBlock = Control as Windows.UI.Xaml.Controls.TextBlock;
                        var shadowLabel = new Label ();
                        shadowLabel.Text = textBlock.Text;
                        shadowLabel.FontAttributes = FontAttributes.Bold;
                        shadowLabel.HorizontalOptions = LayoutOptions.Center;
                        shadowLabel.VerticalOptions = LayoutOptions.CenterAndExpand;
                        shadowLabel.TextColor = effect.Color;
                        shadowLabel.TranslationX = effect.DistanceX;
                        shadowLabel.TranslationY = effect.DistanceY;

                        ((Grid)Element.Parent).Children.Insert (0, shadowLabel);
                        shadowAdded = true;
                    }
                }
            } catch (Exception ex) {
                Debug.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}
```

The Universal Windows Platform doesn't provide a shadow effect, and so the `LabelShadowEffect` implementation on both platforms simulates one by adding a second offset `Label` behind the primary `Label`. The `OnAttached` method retrieves the `ShadowEffect` instance, creates the new `Label`, and sets some layout properties on the `Label`. It then creates the shadow by setting the `TextColor`, `TranslationX`, and `TranslationY` properties to control the color and location of the `Label`. The `shadowLabel` is then inserted offset behind the primary `Label`. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

## Summary

This article has demonstrated using CLR properties to pass parameters to an effect. CLR properties can be used to

define effect parameters that don't respond to runtime property changes.

## Related Links

- [Custom Renderers](#)
- [Effect](#)
- [PlatformEffect](#)
- [RoutingEffect](#)
- [Shadow Effect \(sample\)](#)

# Passing Effect Parameters as Attached Properties

7/12/2018 • 10 minutes to read • [Edit Online](#)

Attached properties can be used to define effect parameters that respond to runtime property changes. This article demonstrates using attached properties to pass parameters to an effect, and changing a parameter at runtime.

The process for creating effect parameters that respond to runtime property changes is as follows:

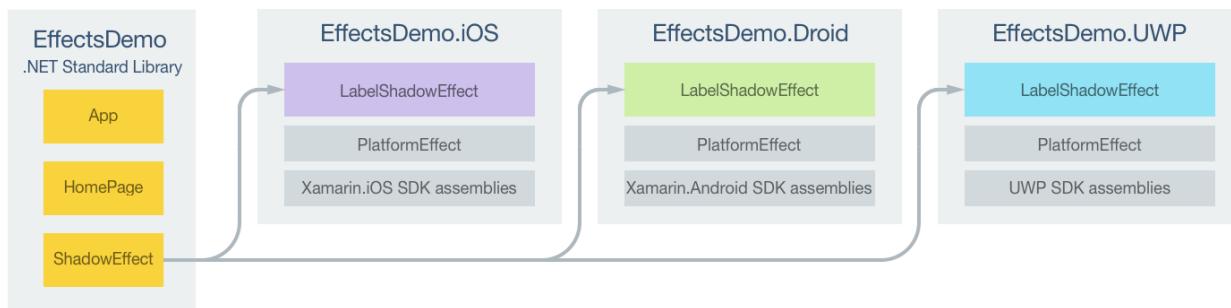
1. Create a `static` class that contains an attached property for each parameter to be passed to the effect.
2. Add an additional attached property to the class that will be used to control the addition or removal of the effect to the control that the class will be attached to. Ensure that this attached property registers a `PropertyChanged` delegate that will be executed when the value of the property changes.
3. Create `static` getters and setters for each attached property.
4. Implement logic in the `PropertyChanged` delegate to add and remove the effect.
5. Implement a nested class inside the `static` class, named after the effect, which subclasses the `RoutingEffect` class. For the constructor, call the base class constructor, passing in a concatenation of the resolution group name, and the unique ID that was specified on each platform-specific effect class.

Parameters can then be passed to the effect by adding the attached properties, and property values, to the appropriate control. In addition, parameters can be changed at runtime by specifying a new attached property value.

## NOTE

An attached property is a special type of bindable property, defined in one class but attached to other objects, and recognizable in XAML as attributes that contain a class and a property name separated by a period. For more information, see [Attached Properties](#).

The sample application demonstrates a `ShadowEffect` that adds a shadow to the text displayed by a `Label` control. In addition, the color of the shadow can be changed at runtime. The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:



A `Label` control on the `HomePage` is customized by the `LabelShadowEffect` in each platform-specific project. Parameters are passed to each `LabelShadowEffect` through attached properties in the `ShadowEffect` class. Each `LabelShadowEffect` class derives from the `PlatformEffect` class for each platform. This results in a shadow being added to the text displayed by the `Label` control, as shown in the following screenshots:

iOS

Android

## Creating Effect Parameters

A `static` class should be created to represent effect parameters, as demonstrated in the following code example:

```
public static class ShadowEffect
{
    public static readonly BindableProperty HasShadowProperty =
        BindableProperty.CreateAttached ("HasShadow", typeof(bool), typeof(ShadowEffect), false, propertyChanged:
OnHasShadowChanged);
    public static readonly BindableProperty ColorProperty =
        BindableProperty.CreateAttached ("Color", typeof(Color), typeof(ShadowEffect), Color.Default);
    public static readonly BindableProperty RadiusProperty =
        BindableProperty.CreateAttached ("Radius", typeof(double), typeof(ShadowEffect), 1.0);
    public static readonly BindableProperty DistanceXProperty =
        BindableProperty.CreateAttached ("DistanceX", typeof(double), typeof(ShadowEffect), 0.0);
    public static readonly BindableProperty DistanceYProperty =
        BindableProperty.CreateAttached ("DistanceY", typeof(double), typeof(ShadowEffect), 0.0);

    public static bool GetHasShadow (BindableObject view)
    {
        return (bool)view.GetValue (HasShadowProperty);
    }

    public static void SetHasShadow (BindableObject view, bool value)
    {
        view.SetValue (HasShadowProperty, value);
    }
    ...

    static void OnHasShadowChanged (BindableObject bindable, object oldValue, object newValue)
    {
        var view = bindable as View;
        if (view == null) {
            return;
        }

        bool hasShadow = (bool)newValue;
        if (hasShadow) {
            view.Effects.Add (new LabelShadowEffect ());
        } else {
            var toRemove = view.Effects.FirstOrDefault (e => e is LabelShadowEffect);
            if (toRemove != null) {
                view.Effects.Remove (toRemove);
            }
        }
    }
}

class LabelShadowEffect : RoutingEffect
{
    public LabelShadowEffect () : base ("MyCompany.LabelShadowEffect")
    {
    }
}
```

The `ShadowEffect` contains five attached properties, with `static` getters and setters for each attached property.

Four of these properties represent parameters to be passed to each platform-specific `LabelShadowEffect`. The `ShadowEffect` class also defines a `HasShadow` attached property that is used to control the addition or removal of the effect to the control that the `ShadowEffect` class is attached to. This attached property registers the `OnHasShadowChanged` method that will be executed when the value of the property changes. This method adds or removes the effect based on the value of the `HasShadow` attached property.

The nested `LabelShadowEffect` class, which subclasses the `RoutingEffect` class, supports effect addition and removal. The `RoutingEffect` class represents a platform-independent effect that wraps an inner effect that is usually platform-specific. This simplifies the effect removal process, since there is no compile-time access to the type information for a platform-specific effect. The `LabelShadowEffect` constructor calls the base class constructor, passing in a parameter consisting of a concatenation of the resolution group name, and the unique ID that was specified on each platform-specific effect class. This enables effect addition and removal in the `OnHasShadowChanged` method, as follows:

- **Effect addition** – a new instance of the `LabelShadowEffect` is added to the control's `Effects` collection. This replaces using the `Effect.Resolve` method to add the effect.
- **Effect removal** – the first instance of the `LabelShadowEffect` in the control's `Effects` collection is retrieved and removed.

## Consuming the Effect

Each platform-specific `LabelShadowEffect` can be consumed by adding the attached properties to a `Label` control, as demonstrated in the following XAML code example:

```
<Label Text="Label Shadow Effect" ...>
    local:ShadowEffect.HasShadow="true" local:ShadowEffect.Radius="5"
    local:ShadowEffect.DistanceX="5" local:ShadowEffect.DistanceY="5">
        <local:ShadowEffect.Color>
            <OnPlatform x:TypeArguments="Color">
                <On Platform="iOS" Value="Black" />
                <On Platform="Android" Value="White" />
                <On Platform="UWP" Value="Red" />
            </OnPlatform>
        </local:ShadowEffect.Color>
    </Label>
```

The equivalent `Label` in C# is shown in the following code example:

```

var label = new Label {
    Text = "Label Shadow Effect",
    ...
};

Color color = Color.Default;
switch (Device.RuntimePlatform)
{
    case Device.iOS:
        color = Color.Black;
        break;
    case Device.Android:
        color = Color.White;
        break;
    case Device.UWP:
        color = Color.Red;
        break;
}

ShadowEffect.SetHasShadow (label, true);
ShadowEffect.SetRadius (label, 5);
ShadowEffect.SetDistanceX (label, 5);
ShadowEffect.SetDistanceY (label, 5);
ShadowEffectSetColor (label, color));

```

Setting the `ShadowEffect.HasShadow` attached property to `true` executes the `ShadowEffect.OnHasShadowChanged` method that adds or removes the `LabelShadowEffect` to the `Label` control. In both code examples, the `ShadowEffect.Color` attached property provides platform-specific color values. For more information, see [Device Class](#).

In addition, a `Button` allows the shadow color to be changed at runtime. When the `Button` is clicked, the following code changes the shadow color by setting the `ShadowEffect.Color` attached property:

```
ShadowEffectSetColor (label, Color.Teal);
```

## Consuming the Effect with a Style

Effects that can be consumed by adding attached properties to a control can also be consumed by a style. The following XAML code example shows an *explicit* style for the shadow effect, that can be applied to `Label` controls:

```

<Style x:Key="ShadowEffectStyle" TargetType="Label">
    <Style.Setters>
        <Setter Property="local:ShadowEffect.HasShadow" Value="True" />
        <Setter Property="local:ShadowEffect.Radius" Value="5" />
        <Setter Property="local:ShadowEffect.DistanceX" Value="5" />
        <Setter Property="local:ShadowEffect.DistanceY" Value="5" />
    </Style.Setters>
</Style>

```

The `Style` can be applied to a `Label` by setting its `Style` property to the `Style` instance using the `StaticResource` markup extension, as demonstrated in the following code example:

```
<Label Text="Label Shadow Effect" ... Style="{StaticResource ShadowEffectStyle}" />
```

For more information about styles, see [Styles](#).

## Creating the Effect on each Platform

The following sections discuss the platform-specific implementation of the `LabelShadowEffect` class.

## iOS Project

The following code example shows the `LabelShadowEffect` implementation for the iOS project:

```
[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                UpdateRadius ();
                UpdateColor ();
                UpdateOffset ();
                Control.Layer.ShadowOpacity = 1.0f;
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateRadius ()
        {
            Control.Layer.CornerRadius = (nfloat)ShadowEffect.GetRadius (Element);
        }

        void UpdateColor ()
        {
            Control.Layer.ShadowColor = ShadowEffect.GetColor (Element).ToCGColor ();
        }

        void UpdateOffset ()
        {
            Control.Layer.ShadowOffset = new CGSize (
                (double)ShadowEffect.GetDistanceX (Element),
                (double)ShadowEffect.GetDistanceY (Element));
        }
    }
}
```

The `OnAttached` method calls methods that retrieve the attached property values using the `ShadowEffect` getters, and which set `Control.Layer` properties to the property values to create the shadow. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

### Responding to Property Changes

If any of the `ShadowEffect` attached property values change at runtime, the effect needs to respond by displaying the changes. An overridden version of the `OnElementPropertyChanged` method, in the platform-specific effect class, is the place to respond to bindable property changes, as demonstrated in the following code example:

```
public class LabelShadowEffect : PlatformEffect
{
    ...
    protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
    {
        if (args.PropertyName == ShadowEffect.RadiusProperty.PropertyName) {
            UpdateRadius ();
        } else if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
            UpdateColor ();
        } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
                   args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
            UpdateOffset ();
        }
    }
    ...
}
```

The `OnElementPropertyChanged` method updates the radius, color, or offset of the shadow, provided that the appropriate `ShadowEffect` attached property value has changed. A check for the property that's changed should always be made, as this override can be called many times.

## Android Project

The following code example shows the `LabelShadowEffect` implementation for the Android project:

```

[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.Droid
{
    public class LabelShadowEffect : PlatformEffect
    {
        Android.Widget.TextView control;
        Android.Graphics.Color color;
        float radius, distanceX, distanceY;

        protected override void OnAttached ()
        {
            try {
                control = Control as Android.Widget.TextView;
                UpdateRadius ();
                UpdateColor ();
                UpdateOffset ();
                UpdateControl ();
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateControl ()
        {
            if (control != null) {
                control.SetShadowLayer (radius, distanceX, distanceY, color);
            }
        }

        void UpdateRadius ()
        {
            radius = (float)ShadowEffect.GetRadius (Element);
        }

        void UpdateColor ()
        {
            color = ShadowEffect.GetColor (Element).ToAndroid ();
        }

        void UpdateOffset ()
        {
            distanceX = (float)ShadowEffect.GetDistanceX (Element);
            distanceY = (float)ShadowEffect.GetDistanceY (Element);
        }
    }
}

```

The `OnAttached` method calls methods that retrieve the attached property values using the `ShadowEffect` getters, and calls a method that calls the `TextView.SetShadowLayer` method to create a shadow using the property values. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

#### Responding to Property Changes

If any of the `ShadowEffect` attached property values change at runtime, the effect needs to respond by displaying the changes. An overridden version of the `OnElementPropertyChanged` method, in the platform-specific effect class, is the place to respond to bindable property changes, as demonstrated in the following code example:

```
public class LabelShadowEffect : PlatformEffect
{
    ...
    protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
    {
        if (args.PropertyName == ShadowEffect.RadiusProperty.PropertyName) {
            UpdateRadius ();
            UpdateControl ();
        } else if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
            UpdateColor ();
            UpdateControl ();
        } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
                   args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
            UpdateOffset ();
            UpdateControl ();
        }
    }
    ...
}
```

The `OnElementPropertyChanged` method updates the radius, color, or offset of the shadow, provided that the appropriate `ShadowEffect` attached property value has changed. A check for the property that's changed should always be made, as this override can be called many times.

### Universal Windows Platform Project

The following code example shows the `LabelShadowEffect` implementation for the Universal Windows Platform (UWP) project:

```

[assembly: ResolutionGroupName ("MyCompany")]
[assembly: ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.UWP
{
    public class LabelShadowEffect : PlatformEffect
    {
        Label shadowLabel;
        bool shadowAdded = false;

        protected override void OnAttached ()
        {
            try {
                if (!shadowAdded) {
                    var textBlock = Control as Windows.UI.Xaml.Controls.TextBlock;

                    shadowLabel = new Label ();
                    shadowLabel.Text = textBlock.Text;
                    shadowLabel.FontAttributes = FontAttributes.Bold;
                    shadowLabel.HorizontalOptions = LayoutOptions.Center;
                    shadowLabel.VerticalOptions = LayoutOptions.CenterAndExpand;

                    UpdateColor ();
                    UpdateOffset ();

                    ((Grid)Element.Parent).Children.Insert (0, shadowLabel);
                    shadowAdded = true;
                }
            } catch (Exception ex) {
                Debug.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateColor ()
        {
            shadowLabel.TextColor = ShadowEffect.GetColor (Element);
        }

        void UpdateOffset ()
        {
            shadowLabel.TranslationX = ShadowEffect.GetDistanceX (Element);
            shadowLabel.TranslationY = ShadowEffect.GetDistanceY (Element);
        }
    }
}

```

The Universal Windows Platform doesn't provide a shadow effect, and so the `LabelShadowEffect` implementation on both platforms simulates one by adding a second offset `Label` behind the primary `Label`. The `OnAttached` method creates the new `Label` and sets some layout properties on the `Label`. It then calls methods that retrieve the attached property values using the `ShadowEffect` getters, and creates the shadow by setting the `TextColor`, `TranslationX`, and `TranslationY` properties to control the color and location of the `Label`. The `shadowLabel` is then inserted offset behind the primary `Label`. This functionality is wrapped in a `try / catch` block in case the control that the effect is attached to does not have the `Control.Layer` properties. No implementation is provided by the `OnDetached` method because no cleanup is necessary.

#### Responding to Property Changes

If any of the `ShadowEffect` attached property values change at runtime, the effect needs to respond by displaying the changes. An overridden version of the `OnElementPropertyChanged` method, in the platform-specific effect class, is the place to respond to bindable property changes, as demonstrated in the following code example:

```
public class LabelShadowEffect : PlatformEffect
{
    ...
    protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
    {
        if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
            UpdateColor ();
        } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
                   args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
            UpdateOffset ();
        }
    }
    ...
}
```

The `OnElementPropertyChanged` method updates the color or offset of the shadow, provided that the appropriate `ShadowEffect` attached property value has changed. A check for the property that's changed should always be made, as this override can be called many times.

## Summary

This article has demonstrated using attached properties to pass parameters to an effect, and changing a parameter at runtime. Attached properties can be used to define effect parameters that respond to runtime property changes.

## Related Links

- [Custom Renderers](#)
- [Effect](#)
- [PlatformEffect](#)
- [RoutingEffect](#)
- [Shadow Effect \(sample\)](#)

# Invoking Events from Effects

11/11/2018 • 21 minutes to read • [Edit Online](#)

An effect can define and invoke an event, signaling changes in the underlying native view. This article shows how to implement low-level multi-touch finger tracking, and how to generate events that signal touch activity.

The effect described in this article provides access to low-level touch events. These low-level events are not available through the existing `GestureRecognizer` classes, but they are vital to some types of applications. For example, a finger-paint application needs to track individual fingers as they move on the screen. A music keyboard needs to detect taps and releases on the individual keys, as well as a finger gliding from one key to another in a glissando.

An effect is ideal for multi-touch finger tracking because it can be attached to any `Xamarin.Forms` element.

## Platform Touch Events

The iOS, Android, and Universal Windows Platform all include a low-level API that allows applications to detect touch activity. These platforms all distinguish between three basic types of touch events:

- *Pressed*, when a finger touches the screen
- *Moved*, when a finger touching the screen moves
- *Released*, when the finger is released from the screen

In a multi-touch environment, multiple fingers can touch the screen at the same time. The various platforms include an identification (ID) number that applications can use to distinguish between multiple fingers.

In iOS, the `UIView` class defines three overridable methods, `TouchesBegan`, `TouchesMoved`, and `TouchesEnded` corresponding to these three basic events. The article [Multi-Touch Finger Tracking](#) describes how to use these methods. However, an iOS program does not need to override a class that derives from `UIView` to use these methods. The iOS `UIGestureRecognizer` also defines these same three methods, and you can attach an instance of a class that derives from `UIGestureRecognizer` to any `UIView` object.

In Android, the `View` class defines an overridable method named `OnTouchEvent` to process all the touch activity. The type of the touch activity is defined by enumeration members `Down`, `PointerDown`, `Move`, `Up`, and `PointerUp` as described in the article [Multi-Touch Finger Tracking](#). The Android `View` also defines an event named `Touch` that allows an event handler to be attached to any `View` object.

In the Universal Windows Platform (UWP), the `UIElement` class defines events named `PointerPressed`, `PointerMoved`, and `PointerReleased`. These are described in the article [Handle Pointer Input article on MSDN](#) and the API documentation for the `UIElement` class.

The `Pointer` API in the Universal Windows Platform is intended to unify mouse, touch, and pen input. For that reason, the `PointerMoved` event is invoked when the mouse moves across an element even when a mouse button is not depressed. The `PointerRoutedEventArgs` object that accompanies these events has a property named `Pointer` that has a property named `IsInContact` which indicates if a mouse button is pressed or a finger is in contact with the screen.

In addition, the UWP defines two more events named `PointerEntered` and `PointerExited`. These indicate when a mouse or finger moves from one element to another. For example, consider two adjacent elements named A and B. Both elements have installed handlers for the pointer events. When a finger presses on A, the `PointerPressed` event is invoked. As the finger moves, A invokes `PointerMoved` events. If the finger moves from A to B, A invokes a

`PointerExited` event and B invokes a `PointerEntered` event. If the finger is then released, B invokes a `PointerReleased` event.

The iOS and Android platforms are different from the UWP: The view that first gets the call to `TouchesBegan` or `OnTouchEvent` when a finger touches the view continues to get all the touch activity even if the finger moves to different views. The UWP can behave similarly if the application captures the pointer: In the `PointerEntered` event handler, the element calls `CapturePointer` and then gets all touch activity from that finger.

The UWP approach proves to be very useful for some types of applications, for example, a music keyboard. Each key can handle the touch events for that key and detect when a finger has slid from one key to another using the `PointerEntered` and `PointerExited` events.

For that reason, the touch-tracking effect described in this article implements the UWP approach.

## The Touch-Tracking Effect API

The **Touch Tracking Effect Demos** sample contains the classes (and an enumeration) that implement the low-level touch-tracking. These types belong to the namespace `TouchTracking` and begin with the word `Touch`. The **TouchTrackingEffectDemos** .NET Standard library project includes the `TouchActionType` enumeration for the type of touch events:

```
public enum TouchActionType
{
    Entered,
    Pressed,
    Moved,
    Released,
    Exited,
    Cancelled
}
```

All the platforms also include an event that indicates that the touch event has been cancelled.

The `TouchEffect` class in the .NET Standard library derives from `RoutingEffect` and defines an event named `TouchAction` and a method named `OnTouchAction` that invokes the `TouchAction` event:

```
public class TouchEffect : RoutingEffect
{
    public event TouchActionEventHandler TouchAction;

    public TouchEffect() : base("XamarinDocs.TouchEffect")
    {
    }

    public bool Capture { set; get; }

    public void OnTouchAction(Element element, TouchActionEventArgs args)
    {
        TouchAction?.Invoke(element, args);
    }
}
```

Also notice the `Capture` property. To capture touch events, an application must set this property to `true` prior to a `Pressed` event. Otherwise, the touch events behave like those in the Universal Windows Platform.

The `TouchActionEventArgs` class in the .NET Standard library contains all the information that accompanies each event:

```

public class TouchEventArgs : EventArgs
{
    public TouchEventArgs(long id, TouchActionType type, Point location, bool isInContact)
    {
        Id = id;
        Type = type;
        Location = location;
        IsInContact = isInContact;
    }

    public long Id { private set; get; }

    public TouchActionType Type { private set; get; }

    public Point Location { private set; get; }

    public bool IsInContact { private set; get; }
}

```

An application can use the `Id` property for tracking individual fingers. Notice the `IsInContact` property. This property is always `true` for `Pressed` events and `false` for `Released` events. It's also always `true` for `Moved` events on iOS and Android. The `IsInContact` property might be `false` for `Moved` events on the Universal Windows Platform when the program is running on the desktop and the mouse pointer moves without a button pressed.

You can use the `TouchEffect` class in your own applications by including the file in the solution's .NET Standard library project, and by adding an instance to the `Effects` collection of any `Xamarin.Forms` element. Attach a handler to the `TouchAction` event to obtain the touch events.

To use `TouchEffect` in your own application, you'll also need the platform implementations included in **TouchTrackingEffectDemos** solution.

## The Touch-Tracking Effect Implementations

The iOS, Android, and UWP implementations of the `TouchEffect` are described below beginning with the simplest implementation (UWP) and ending with the iOS implementation because it is more structurally complex than the others.

### The UWP Implementation

The UWP implementation of `TouchEffect` is the simplest. As usual, the class derives from `PlatformEffect` and includes two assembly attributes:

```

[assembly: ResolutionGroupName("XamarinDocs")]
[assembly: ExportEffect(typeof(TouchTracking.UWP.TouchEffect), "TouchEffect")]

namespace TouchTracking.UWP
{
    public class TouchEffect : PlatformEffect
    {
        ...
    }
}

```

The `OnAttached` override saves some information as fields and attaches handlers to all the pointer events:

```

public class TouchEffect : PlatformEffect
{
    FrameworkElement frameworkElement;
    TouchTracking.TouchEffect effect;
    Action<Element, TouchActionEventArgs> onTouchAction;

    protected override void OnAttached()
    {
        // Get the Windows FrameworkElement corresponding to the Element that the effect is attached to
        frameworkElement = Control == null ? Container : Control;

        // Get access to the TouchEffect class in the .NET Standard library
        effect = (TouchTracking.TouchEffect)Element.Effects.
            FirstOrDefault(e => e is TouchTracking.TouchEffect);

        if (effect != null && frameworkElement != null)
        {
            // Save the method to call on touch events
            onTouchAction = effect.OnTouchAction;

            // Set event handlers on FrameworkElement
            frameworkElement.PointerEntered += OnPointerEntered;
            frameworkElement.PointerPressed += OnPointerPressed;
            frameworkElement.PointerMoved += OnPointerMoved;
            frameworkElement.PointerReleased += OnPointerReleased;
            frameworkElement.PointerExited += OnPointerExited;
            frameworkElement.PointerCancelled += OnPointerCancelled;
        }
    }
}
...

```

The `OnPointerPressed` handler invokes the effect event by calling the `onTouchAction` field in the `CommonHandler` method:

```

public class TouchEffect : PlatformEffect
{
    ...
    void OnPointerPressed(object sender, PointerRoutedEventArgs args)
    {
        CommonHandler(sender, TouchActionType.Pressed, args);

        // Check setting of Capture property
        if (effect.Capture)
        {
            (sender as FrameworkElement).CapturePointer(args.Pointer);
        }
    }
    ...
    void CommonHandler(object sender, TouchActionType touchActionType, PointerRoutedEventArgs args)
    {
        PointerPoint pointerPoint = args.GetCurrentPoint(sender as UIElement);
        Windows.Foundation.Point windowsPoint = pointerPoint.Position;

        onTouchAction(Element, new TouchActionEventArgs(args.Pointer.PointerId,
                                                        touchActionType,
                                                        new Point(windowsPoint.X, windowsPoint.Y),
                                                        args.Pointer.IsInContact));
    }
}

```

`OnPointerPressed` also checks the value of the `Capture` property in the effect class in the .NET Standard library and calls `CapturePointer` if it is `true`.

The other UWP event handlers are even simpler:

```
public class TouchEffect : PlatformEffect
{
    ...
    void OnPointerEntered(object sender, PointerRoutedEventArgs args)
    {
        CommonHandler(sender, TouchActionType.Entered, args);
    }
    ...
}
```

## The Android Implementation

The Android and iOS implementations are necessarily more complex because they must implement the `Exited` and `Entered` events when a finger moves from one element to another. Both implementations are structured similarly.

The Android `TouchEffect` class installs a handler for the `Touch` event:

```
view = Control == null ? Container : Control;
...
view.Touch += OnTouch;
```

The class also defines two static dictionaries:

```
public class TouchEffect : PlatformEffect
{
    ...
    static Dictionary<Android.Views.View, TouchEffect> viewDictionary =
        new Dictionary<Android.Views.View, TouchEffect>();

    static Dictionary<int, TouchEffect> idToEffectDictionary =
        new Dictionary<int, TouchEffect>();
    ...
}
```

The `viewDictionary` gets a new entry every time the `OnAttached` override is called:

```
viewDictionary.Add(view, this);
```

The entry is removed from the dictionary in `OnDetached`. Every instance of `TouchEffect` is associated with a particular view that the effect is attached to. The static dictionary allows any `TouchEffect` instance to enumerate through all the other views and their corresponding `TouchEffect` instances. This is necessary to allow for transferring the events from one view to another.

Android assigns an ID code to touch events that allows an application to track individual fingers. The `idToEffectDictionary` associates this ID code with a `TouchEffect` instance. An item is added to this dictionary when the `Touch` handler is called for a finger press:

```

void OnTouch(object sender, Android.Views.View.TouchEventArgs args)
{
    ...
    switch (args.Event.ActionMasked)
    {
        case MotionEventActions.Down:
        case MotionEventActions.PointerDown:
            FireEvent(this, id, TouchActionType.Pressed, screenPointerCoords, true);

            idToEffectDictionary.Add(id, this);

            capture = libTouchEffect.Capture;
            break;
    }
}

```

The item is removed from the `idToEffectDictionary` when the finger is released from the screen. The `FireEvent` method simply accumulates all the information necessary to call the `OnTouchAction` method:

```

void FireEvent(TouchEffect touchEffect, int id, TouchActionType actionType, Point pointerLocation, bool
isInContact)
{
    // Get the method to call for firing events
    Action<Element, TouchEventArgs> onTouchAction = touchEffect.libTouchEffect.OnTouchAction;

    // Get the location of the pointer within the view
    touchEffect.view.GetLocationOnScreen(twoIntArray);
    double x = pointerLocation.X - twoIntArray[0];
    double y = pointerLocation.Y - twoIntArray[1];
    Point point = new Point(fromPixels(x), fromPixels(y));

    // Call the method
    onTouchAction(touchEffect.formsElement,
        new TouchEventArgs(id, actionType, point, isInContact));
}

```

All the other touch types are processed in two different ways: If the `Capture` property is `true`, the touch event is a fairly simple translation to the `TouchEffect` information. It gets more complicated when `Capture` is `false` because the touch events might need to be moved from one view to another. This is the responsibility of the `CheckForBoundaryHop` method, which is called during move events. This method makes use of both static dictionaries. It enumerates through the `viewDictionary` to determine the view that the finger is currently touching, and it uses `idToEffectDictionary` to store the current `TouchEffect` instance (and hence, the current view) associated with a particular ID:

```

void CheckForBoundaryHop(int id, Point pointerLocation)
{
    TouchEffect touchEffectHit = null;

    foreach (Android.Views.View view in viewDictionary.Keys)
    {
        // Get the view rectangle
        try
        {
            view.GetLocationOnScreen(twoIntArray);
        }
        catch // System.ObjectDisposedException: Cannot access a disposed object.
        {
            continue;
        }
        Rectangle viewRect = new Rectangle(twoIntArray[0], twoIntArray[1], view.Width, view.Height);

        if (viewRect.Contains(pointerLocation))
        {
            touchEffectHit = viewDictionary[view];
        }
    }

    if (touchEffectHit != idToEffectDictionary[id])
    {
        if (idToEffectDictionary[id] != null)
        {
            FireEvent(idToEffectDictionary[id], id, TouchActionType.Exited, pointerLocation, true);
        }
        if (touchEffectHit != null)
        {
            FireEvent(touchEffectHit, id, TouchActionType.Entered, pointerLocation, true);
        }
        idToEffectDictionary[id] = touchEffectHit;
    }
}

```

If there's been a change in the `idToEffectDictionary`, the method potentially calls `FireEvent` for `Exited` and `Entered` to transfer from one view to another. However, the finger might have been moved to an area occupied by a view without an attached `TouchEffect`, or from that area to a view with the effect attached.

Notice the `try` and `catch` block when the view is accessed. In a page that is navigated to that then navigates back to the home page, the `OnDetached` method is not called and items remain in the `viewDictionary` but Android considers them disposed.

### The iOS Implementation

The iOS implementation is similar to the Android implementation except that the iOS `TouchEffect` class must instantiate a derivative of `UIGestureRecognizer`. This is a class in the iOS project named `TouchRecognizer`. This class maintains two static dictionaries that store `TouchRecognizer` instances:

```

static Dictionary<UIView, TouchRecognizer> viewDictionary =
    new Dictionary<UIView, TouchRecognizer>();

static Dictionary<long, TouchRecognizer> idToTouchDictionary =
    new Dictionary<long, TouchRecognizer>();

```

Much of the structure of this `TouchRecognizer` class is similar to the Android `TouchEffect` class.

## Putting the Touch Effect to Work

The **TouchTrackingEffectDemos** program contains five pages that test the touch-tracking effect for common

tasks.

The **BoxView Dragging** page allows you to add `BoxView` elements to an `AbsoluteLayout` and then drag them around the screen. The [XAML file](#) instantiates two `Button` views for adding `BoxView` elements to the `AbsoluteLayout` and clearing the `AbsoluteLayout`.

The method in the [code-behind file](#) that adds a new `BoxView` to the `AbsoluteLayout` also adds a `TouchEffect` object to the `BoxView` and attaches an event handler to the effect:

```
void AddBoxViewToLayout()
{
    BoxView boxView = new BoxView
    {
        WidthRequest = 100,
        HeightRequest = 100,
        Color = new Color(random.NextDouble(),
                          random.NextDouble(),
                          random.NextDouble())
    };

    TouchEffect touchEffect = new TouchEffect();
    touchEffect.TouchAction += OnTouchEffectAction;
    boxView.Effects.Add(touchEffect);
    absoluteLayout.Children.Add(boxView);
}
```

The `TouchAction` event handler processes all the touch events for all the `BoxView` elements, but it needs to exercise some caution: It can't allow two fingers on a single `BoxView` because the program only implements dragging, and the two fingers would interfere with each other. For this reason, the page defines an embedded class for each finger currently being tracked:

```
class DragInfo
{
    public DragInfo(long id, Point pressPoint)
    {
        Id = id;
        PressPoint = pressPoint;
    }

    public long Id { private set; get; }

    public Point PressPoint { private set; get; }
}

Dictionary<BoxView, DragInfo> dragDictionary = new Dictionary<BoxView, DragInfo>();
```

The `dragDictionary` contains an entry for every `BoxView` currently being dragged.

The `Pressed` touch action adds an item to this dictionary, and the `Released` action removes it. The `Pressed` logic must check if there's already an item in the dictionary for that `BoxView`. If so, the `BoxView` is already being dragged and the new event is a second finger on that same `BoxView`. For the `Moved` and `Released` actions, the event handler must check if the dictionary has an entry for that `BoxView` and that the touch `Id` property for that dragged `BoxView` matches the one in the dictionary entry:

```

void OnTouchEffectAction(object sender, TouchEventArgs args)
{
    BoxView boxView = sender as BoxView;

    switch (args.Type)
    {
        case TouchActionType.Pressed:
            // Don't allow a second touch on an already touched BoxView
            if (!dragDictionary.ContainsKey(boxView))
            {
                dragDictionary.Add(boxView, new DragInfo(args.Id, args.Location));

                // Set Capture property to true
                TouchEffect touchEffect = (TouchEffect)boxView.Effects.FirstOrDefault(e => e is TouchEffect);
                touchEffect.Capture = true;
            }
            break;

        case TouchActionType.Moved:
            if (dragDictionary.ContainsKey(boxView) && dragDictionary[boxView].Id == args.Id)
            {
                Rectangle rect = AbsoluteLayout.GetLayoutBounds(boxView);
                Point initialLocation = dragDictionary[boxView].PressPoint;
                rect.X += args.Location.X - initialLocation.X;
                rect.Y += args.Location.Y - initialLocation.Y;
                AbsoluteLayout.SetLayoutBounds(boxView, rect);
            }
            break;

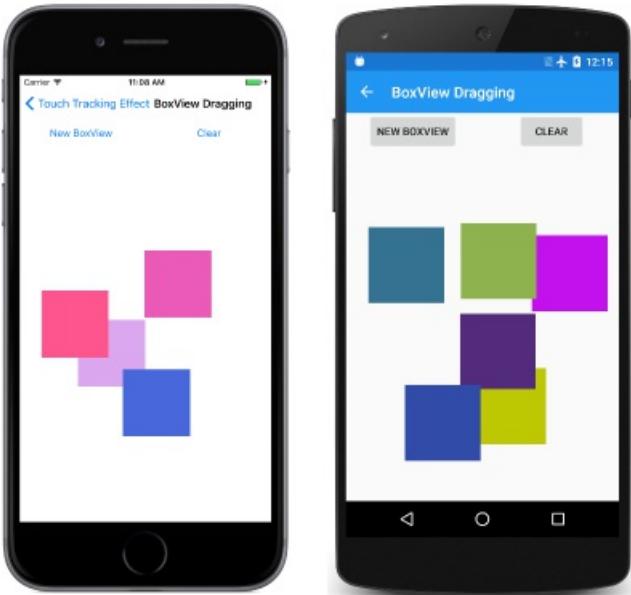
        case TouchActionType.Released:
            if (dragDictionary.ContainsKey(boxView) && dragDictionary[boxView].Id == args.Id)
            {
                dragDictionary.Remove(boxView);
            }
            break;
    }
}

```

The `Pressed` logic sets the `Capture` property of the `TouchEffect` object to `true`. This has the effect of delivering all subsequent events for that finger to the same event handler.

The `Moved` logic moves the `BoxView` by altering the `LayoutBounds` attached property. The `Location` property of the event arguments is always relative to the `BoxView` being dragged, and if the `BoxView` is being dragged at a constant rate, the `Location` properties of the consecutive events will be approximately the same. For example, if a finger presses the `BoxView` in its center, the `Pressed` action stores a `PressPoint` property of (50, 50), which remains the same for subsequent events. If the `BoxView` is dragged diagonally at a constant rate, the subsequent `Location` properties during the `Moved` action might be values of (55, 55), in which case the `Moved` logic adds 5 to the horizontal and vertical position of the `BoxView`. This moves the `BoxView` so that its center is again directly under the finger.

You can move multiple `BoxView` elements simultaneously using different fingers.



### Subclassing the View

Often, it's easier for a Xamarin.Forms element to handle its own touch events. The **Draggable BoxView Dragging** page functions the same as the **BoxView Dragging** page, but the elements that the user drags are instances of a `DraggableBoxView` class that derives from `BoxView`:

```

class DraggableBoxView : BoxView
{
    bool isBeingDragged;
    long touchId;
    Point pressPoint;

    public DraggableBoxView()
    {
        TouchEffect touchEffect = new TouchEffect
        {
            Capture = true
        };
        touchEffect.TouchAction += OnTouchEffectAction;
        Effects.Add(touchEffect);
    }

    void OnTouchEffectAction(object sender, TouchEventArgs args)
    {
        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (!isBeingDragged)
                {
                    isBeingDragged = true;
                    touchId = args.Id;
                    pressPoint = args.Location;
                }
                break;

            case TouchActionType.Moved:
                if (isBeingDragged && touchId == args.Id)
                {
                    TranslationX += args.Location.X - pressPoint.X;
                    TranslationY += args.Location.Y - pressPoint.Y;
                }
                break;

            case TouchActionType.Released:
                if (isBeingDragged && touchId == args.Id)
                {
                    isBeingDragged = false;
                }
                break;
        }
    }
}

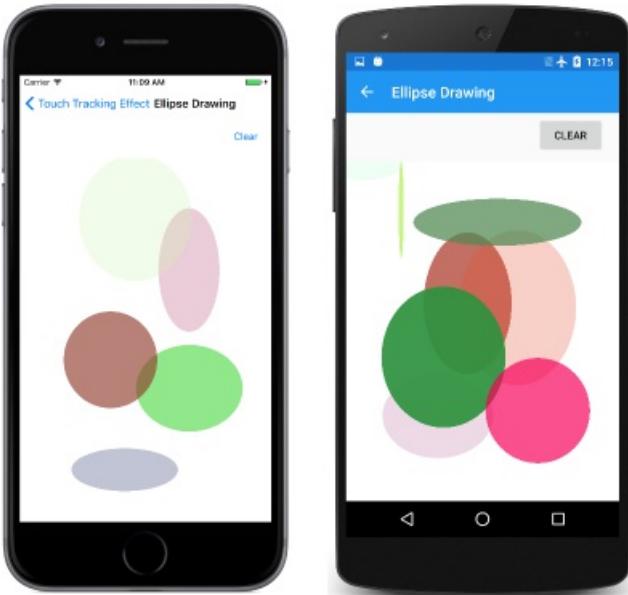
```

The constructor creates and attaches the `TouchEffect`, and sets the `Capture` property when that object is first instantiated. No dictionary is required because the class itself stores `isBeingDragged`, `pressPoint`, and `touchId` values associated with each finger. The `Moved` handling alters the `TranslationX` and `TranslationY` properties so the logic will work even if the parent of the `DraggableBoxView` is not an `AbsoluteLayout`.

## Integrating with SkiaSharp

The next two demonstrations require graphics, and they use SkiaSharp for this purpose. You might want to learn about [Using SkiaSharp in Xamarin.Forms](#) before you study these examples. The first two articles ("SkiaSharp Drawing Basics" and "SkiaSharp Lines and Paths") cover everything that you'll need here.

The **Ellipse Drawing** page allows you to draw an ellipse by swiping your finger on the screen. Depending how you move your finger, you can draw the ellipse from the upper-left to the lower-right, or from any other corner to the opposite corner. The ellipse is drawn with a random color and opacity.



If you then touch one of the ellipses, you can drag it to another location. This requires a technique known as "hit-testing," which involves searching for the graphical object at a particular point. The SkiaSharp ellipses are not Xamarin.Forms elements, so they cannot perform their own `TouchEffect` processing. The `TouchEffect` must apply to the entire `SKCanvasView` object.

The `EllipseDrawPage.xaml` file instantiates the `SKCanvasView` in a single-cell `Grid`. The `TouchEffect` object is attached to that `Grid`:

```
<Grid x:Name="canvasViewGrid"
      Grid.Row="1"
      BackgroundColor="White">

    <skia:SKCanvasView x:Name="canvasView"
                      PaintSurface="OnCanvasViewPaintSurface" />
    <Grid.Effects>
        <tt:TouchEffect Capture="True"
                        TouchAction="OnTouchEventAction" />
    </Grid.Effects>
</Grid>
```

In Android and the Universal Windows Platform, the `TouchEffect` can be attached directly to the `SKCanvasView`, but on iOS that doesn't work. Notice that the `Capture` property is set to `true`.

Each ellipse that SkiaSharp renders is represented by an object of type `EllipseDrawingFigure`:

```

class EllipseDrawingFigure
{
    SKPoint pt1, pt2;

    public EllipseDrawingFigure()
    {
    }

    public SKColor Color { set; get; }

    public SKPoint StartPoint
    {
        set
        {
            pt1 = value;
            MakeRectangle();
        }
    }

    public SKPoint EndPoint
    {
        set
        {
            pt2 = value;
            MakeRectangle();
        }
    }

    void MakeRectangle()
    {
        Rectangle = new SKRect(pt1.X, pt1.Y, pt2.X, pt2.Y).Standardized;
    }

    public SKRect Rectangle { set; get; }

    // For dragging operations
    public Point LastFingerLocation { set; get; }

    // For the dragging hit-test
    public bool IsInEllipse(SKPoint pt)
    {
        SKRect rect = Rectangle;

        return (Math.Pow(pt.X - rect.MidX, 2) / Math.Pow(rect.Width / 2, 2) +
               Math.Pow(pt.Y - rect.MidY, 2) / Math.Pow(rect.Height / 2, 2)) < 1;
    }
}

```

The `StartPoint` and `EndPoint` properties are used when the program is processing touch input; the `Rectangle` property is used for drawing the ellipse. The `LastFingerLocation` property comes into play when the ellipse is being dragged, and the `IsInEllipse` method aids in hit-testing. The method returns `true` if the point is inside the ellipse.

The [code-behind file](#) maintains three collections:

```

Dictionary<long, EllipseDrawingFigure> inProgressFigures = new Dictionary<long, EllipseDrawingFigure>();
List<EllipseDrawingFigure> completedFigures = new List<EllipseDrawingFigure>();
Dictionary<long, EllipseDrawingFigure> draggingFigures = new Dictionary<long, EllipseDrawingFigure>();

```

The `draggingFigure` dictionary contains a subset of the `completedFigures` collection. The SkiaSharp `PaintSurface` event handler simply renders the objects in these the `completedFigures` and `inProgressFigures` collections:

```
SKPaint paint = new SKPaint
{
    Style = SKPaintStyle.Fill
};
...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKCanvas canvas = args.Surface.Canvas;
    canvas.Clear();

    foreach (EllipseDrawingFigure figure in completedFigures)
    {
        paint.Color = figure.Color;
        canvas.DrawOval(figure.Rectangle, paint);
    }
    foreach (EllipseDrawingFigure figure in inProgressFigures.Values)
    {
        paint.Color = figure.Color;
        canvas.DrawOval(figure.Rectangle, paint);
    }
}
```

The trickiest part of the touch processing is the `Pressed` handling. This is where the hit-testing is performed, but if the code detects an ellipse under the user's finger, that ellipse can only be dragged if it's not currently being dragged by another finger. If there is no ellipse under the user's finger, then the code begins the process of drawing a new ellipse:

```

case TouchActionType.Pressed:
    bool isDragOperation = false;

    // Loop through the completed figures
    foreach (EllipseDrawingFigure fig in completedFigures.Reverse<EllipseDrawingFigure>())
    {
        // Check if the finger is touching one of the ellipses
        if (fig.IsInEllipse(ConvertToPixel(args.Location)))
        {
            // Tentatively assume this is a dragging operation
            isDragOperation = true;

            // Loop through all the figures currently being dragged
            foreach (EllipseDrawingFigure draggedFigure in draggingFigures.Values)
            {
                // If there's a match, we'll need to dig deeper
                if (fig == draggedFigure)
                {
                    isDragOperation = false;
                    break;
                }
            }

            if (isDragOperation)
            {
                fig.LastFingerLocation = args.Location;
                draggingFigures.Add(args.Id, fig);
                break;
            }
        }
    }

    if (isDragOperation)
    {
        // Move the dragged ellipse to the end of completedFigures so it's drawn on top
        EllipseDrawingFigure fig = draggingFigures[args.Id];
        completedFigures.Remove(fig);
        completedFigures.Add(fig);
    }
    else // start making a new ellipse
    {
        // Random bytes for random color
        byte[] buffer = new byte[4];
        random.NextBytes(buffer);

        EllipseDrawingFigure figure = new EllipseDrawingFigure
        {
            Color = new SKColor(buffer[0], buffer[1], buffer[2], buffer[3]),
            StartPoint = ConvertToPixel(args.Location),
            EndPoint = ConvertToPixel(args.Location)
        };
        inProgressFigures.Add(args.Id, figure);
    }
    canvasView.InvalidateSurface();
    break;
}

```

The other SkiaSharp example is the **Finger Paint** page. You can select a stroke color and stroke width from two **Picker** views and then draw with one or more fingers:



This example also requires a separate class to represent each line painted on the screen:

```
class FingerPaintPolyline
{
    public FingerPaintPolyline()
    {
        Path = new SKPath();
    }

    public SKPath Path { set; get; }

    public Color StrokeColor { set; get; }

    public float StrokeWidth { set; get; }
}
```

An `SKPath` object is used to render each line. The `FingerPaint.xaml.cs` file maintains two collections of these objects, one for those polylines currently being drawn and another for the completed polylines:

```
Dictionary<long, FingerPaintPolyline> inProgressPolylines = new Dictionary<long, FingerPaintPolyline>();
List<FingerPaintPolyline> completedPolylines = new List<FingerPaintPolyline>();
```

The `Pressed` processing creates a new `FingerPaintPolyline`, calls `MoveTo` on the path object to store the initial point, and adds that object to the `inProgressPolylines` dictionary. The `Moved` processing calls `LineTo` on the path object with the new finger position, and the `Released` processing transfers the completed polyline from `inProgressPolylines` to `completedPolylines`. Once again, the actual SkiaSharp drawing code is relatively simple:

```

SKPaint paint = new SKPaint
{
    Style = SKPaintStyle.Stroke,
    StrokeCap = SKStrokeCap.Round,
    StrokeJoin = SKStrokeJoin.Round
};

...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKCanvas canvas = args.Surface.Canvas;
    canvas.Clear();

    foreach (FingerPaintPolyline polyline in completedPolylines)
    {
        paint.Color = polyline.StrokeColor.ToSKColor();
        paint.StrokeWidth = polyline.StrokeWidth;
        canvas.DrawPath(polyline.Path, paint);
    }

    foreach (FingerPaintPolyline polyline in inProgressPolylines.Values)
    {
        paint.Color = polyline.StrokeColor.ToSKColor();
        paint.StrokeWidth = polyline.StrokeWidth;
        canvas.DrawPath(polyline.Path, paint);
    }
}

```

## Tracking View-to-View Touch

All the previous examples have set the `Capture` property of the `TouchEffect` to `true`, either when the `TouchEffect` was created or when the `Pressed` event occurred. This ensures that the same element receives all the events associated with the finger that first pressed the view. The final sample does *not* set `Capture` to `true`. This causes different behavior when a finger in contact with the screen moves from one element to another. The element that the finger moves from receives an event with a `Type` property set to `TouchActionType.Exited` and the second element receives an event with a `Type` setting of `TouchActionType.Entered`.

This type of touch processing is very useful for a music keyboard. A key should be able to detect when it's pressed, but also when a finger slides from one key to another.

The **Silent Keyboard** page defines small `WhiteKey` and `BlackKey` classes that derive from `Key`, which derives from `BoxView`.

The `Key` class is ready to be used in an actual music program. It defines public properties named `IsPressed` and `KeyNumber`, which is intended to be set to the key code established by the MIDI standard. The `Key` class also defines an event named `StatusChanged`, which is invoked when the `IsPressed` property changes.

Multiple fingers are allowed on each key. For this reason, the `Key` class maintains a `List` of the touch ID numbers of all the fingers currently touching that key:

```
List<long> ids = new List<long>();
```

The `TouchAction` event handler adds an ID to the `ids` list for both a `Pressed` event type and an `Entered` type, but only when the `IsInContact` property is `true` for the `Entered` event. The ID is removed from the `List` for a `Released` or `Exited` event:

```

void OnTouchEffectAction(object sender, TouchEventArgs args)
{
    switch (args.Type)
    {
        case TouchActionType.Pressed:
            AddToList(args.Id);
            break;

        case TouchActionType.Entered:
            if (args.IsInContact)
            {
                AddToList(args.Id);
            }
            break;

        case TouchActionType.Moved:
            break;

        case TouchActionType.Released:
        case TouchActionType.Exited:
            RemoveFromList(args.Id);
            break;
    }
}

```

The `AddToList` and `RemoveFromList` methods both check if the `List` has changed between empty and non-empty, and if so, invokes the `StatusChanged` event.

The various `WhiteKey` and `BlackKey` elements are arranged in the page's [XAML file](#), which looks best when the phone is held in a landscape mode:



If you sweep your finger across the keys, you'll see by the slight changes in color that the touch events are transferred from one key to another.

## Summary

This article has demonstrated how to invoke events in an effect, and how to write and use an effect that implements low-level multi-touch processing.

## Related Links

- Multi-Touch Finger Tracking in iOS
- Multi-Touch Finger Tracking in Android
- Touch Tracking Effect (sample)

# File Handling in Xamarin.Forms

11/11/2018 • 4 minutes to read • [Edit Online](#)

*File handling with Xamarin.Forms can be achieved using code in a .NET Standard library, or by using embedded resources.*

## Overview

Xamarin.Forms code runs on multiple platforms - each of which has its own filesystem. Previously, this meant that reading and writing files was most easily performed using the native file APIs on each platform. Alternatively, embedded resources are a simpler solution to distribute data files with an app. However, with .NET Standard 2.0 it's possible to share file access code in .NET Standard libraries.

For information on handling image files, refer to the [Working with Images](#) page.

## Saving and Loading Files

The `System.IO` classes can be used to access the file system on each platform. The `File` class lets you create, delete, and read files, and the `Directory` class allows you to create, delete, or enumerate the contents of directories. You can also use the `Stream` subclasses, which can provide a greater degree of control over file operations (such as compression or position search within a file).

A text file can be written using the `File.WriteAllText` method:

```
File.WriteAllText(fileName, text);
```

A text file can be read using the `File.ReadAllText` method:

```
string text = File.ReadAllText(fileName);
```

In addition, the `File.Exists` method determines whether the specified file exists:

```
bool doesExist = File.Exists(fileName);
```

The path of the file on each platform can be determined from a .NET Standard library by using a value of the `Environment.SpecialFolder` enumeration as the first argument to the `Environment.GetFolderPath` method. This can then be combined with a filename with the `Path.Combine` method:

```
string fileName = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),  
    "temp.txt");
```

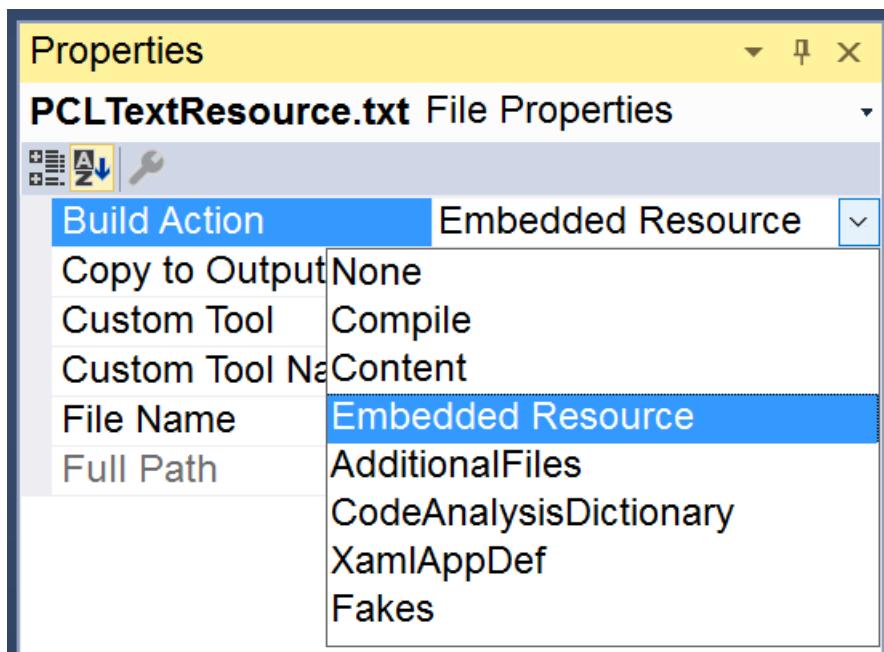
These operations are demonstrated in the sample app, which includes a page that saves and loads text:



## Loading Files Embedded as Resources

To embed a file into a **.NET Standard** assembly, create or add a file and ensure that **Build Action:** **EmbeddedResource**.

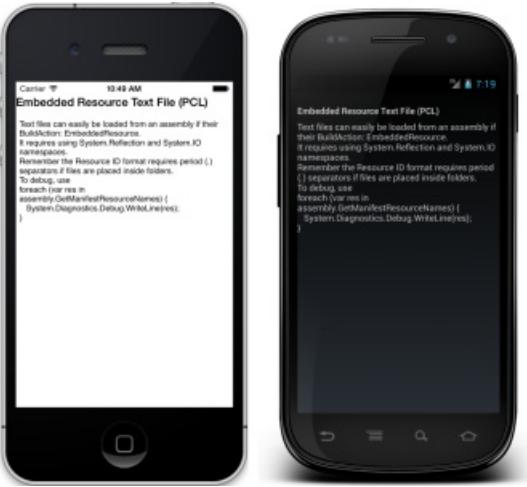
- [Visual Studio](#)
- [Visual Studio for Mac](#)



`GetManifestResourceStream` is used to access the embedded file using its **Resource ID**. By default the resource ID is the filename prefixed with the default namespace for the project it is embedded in - in this case the assembly is **WorkingWithFiles** and the filename is **PCLTextResource.txt**, so the resource ID is `WorkingWithFiles.PCLTextResource.txt`.

```
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(LoadResourceText)).Assembly;
Stream stream = assembly.GetManifestResourceStream("WorkingWithFiles.PCLTextResource.txt");
string text = "";
using (var reader = new System.IO.StreamReader (stream)) {
    text = reader.ReadToEnd ();
}
```

The `text` variable can then be used to display the text or otherwise use it in code. This screenshot of the [sample app](#) shows the text rendered in a `Label` control.



Loading and deserializing an XML is equally simple. The following code shows an XML file being loaded and deserialized from a resource, then bound to a `ListView` for display. The XML file contains an array of `Monkey` objects (the class is defined in the sample code).

```
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(LoadResourceText)).Assembly;
Stream stream = assembly.GetManifestResourceStream("WorkingWithFiles.PCLXmlResource.xml");
List<Monkey> monkeys;
using (var reader = new System.IO.StreamReader (stream)) {
    var serializer = new XmlSerializer(typeof(List<Monkey>));
    monkeys = (List<Monkey>)serializer.Deserialize(reader);
}
var listView = new ListView ();
listView.ItemsSource = monkeys;
```



## Embedding in Shared Projects

Shared Projects can also contain files as embedded resources, however because the contents of a Shared Project are compiled into the referencing projects, the prefix used for embedded file resource IDs can change. This means the resource ID for each embedded file may be different for each platform.

There are two solutions to this issue with Shared Projects:

- **Synchronize the Projects** - Edit the project properties for each platform to use the **same** assembly name and default namespace. This value can then be "hardcoded" as the prefix for embedded resource IDs in the Shared Project.
- **#if compiler directives** - Use compiler directives to set the correct resource ID prefix and use that value to dynamically construct the correct resource ID.

Code illustrating the second option is shown below. Compiler directives are used to select the hardcoded resource prefix (which is normally the same as the default namespace for the referencing project). The `resourcePrefix` variable is then used to create a valid resource ID by concatenating it with the embedded resource filename.

```
#if __IOS__
var resourcePrefix = "WorkingWithFiles.iOS.";
#endif
#if __ANDROID__
var resourcePrefix = "WorkingWithFiles.Droid.";
#endif

Debug.WriteLine("Using this resource prefix: " + resourcePrefix);
// note that the prefix includes the trailing period '.' that is required
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(SharedPage)).Assembly;
Stream stream = assembly.GetManifestResourceStream
    (resourcePrefix + "SharedTextResource.txt");
```

## Organizing Resources

The above examples assume that the file is embedded in the root of the .NET Standard library project, in which case the resource ID is of the form **Namespace.Filename.Extension**, such as

`WorkingWithFiles.PCLTextResource.txt` and `WorkingWithFiles.iOS.SharedTextResource.txt`.

It is possible to organize embedded resources in folders. When an embedded resource is placed in a folder, the folder name becomes part of the resource ID (separated by periods), so that the resource ID format becomes

**Namespace.Folder.Filename.Extension**. Placing the files used in the sample app into a folder **MyFolder** would make the corresponding resource IDs `WorkingWithFiles.MyFolder.PCLTextResource.txt` and

`WorkingWithFiles.iOS.MyFolder.SharedTextResource.txt`.

## Debugging Embedded Resources

Because it is sometimes difficult to understand why a particular resource isn't being loaded, the following debug code can be added temporarily to an application to help confirm the resources are correctly configured. It will output all known resources embedded in the given assembly to the **Errors** pad to help debug resource loading issues.

```
using System.Reflection;
// ...
// use for debugging, not in released app code!
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(SharedPage)).Assembly;
foreach (var res in assembly.GetManifestResourceNames()) {
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}
```

## Summary

This article has shown some simple file operations for saving and loading text on the device, and for loading embedded resources. With .NET Standard 2.0 it's possible to share file access code in .NET Standard libraries.

## Related Links

- [FileSample](#)
- [Xamarin.Forms Samples](#)
- [Working with the File System in Xamarin.iOS](#)

# Xamarin.Forms gestures

9/20/2018 • 2 minutes to read • [Edit Online](#)

*Gesture recognizers can be used to detect user interaction with views in a Xamarin.Forms application.*

The Xamarin.Forms [GestureRecognizer](#) class supports tap, pinch, pan, and swipe gestures on [View](#) instances.

## Adding a tap gesture recognizer

A tap gesture is used for tap detection and is recognized with the [TapGestureRecognizer](#) class.

## Adding a pinch gesture recognizer

A pinch gesture is used for performing interactive zoom and is recognized with the [PinchGestureRecognizer](#) class.

## Adding a pan gesture recognizer

A pan gesture is used for detecting the movement of fingers around the screen and applying that movement to content, and is recognized with the [PanGestureRecognizer](#) class.

## Adding a swipe gesture recognizer

A swipe gesture occurs when a finger is moved across the screen in a horizontal or vertical direction, and is often used to initiate navigation through content. Swipe gestures are recognized with the [SwipeGestureRecognizer](#) class.

# Adding a tap gesture recognizer

9/20/2018 • 2 minutes to read • [Edit Online](#)

The tap gesture is used for tap detection and is implemented with the `TapGestureRecognizer` class.

To make a user interface element clickable with the tap gesture, create a `TapGestureRecognizer` instance, handle the `Tapped` event and add the new gesture recognizer to the `GestureRecognizers` collection on the user interface element. The following code example shows a `TapGestureRecognizer` attached to an `Image` element:

```
var tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.Tapped += (s, e) => {
    // handle the tap
};
image.GestureRecognizers.Add(tapGestureRecognizer);
```

By default the image will respond to single taps. Set the `NumberOfTapsRequired` property to wait for a double-tap (or more taps if required).

```
tapGestureRecognizer.NumberOfTapsRequired = 2; // double-tap
```

When `NumberOfTapsRequired` is set above one, the event handler will only be executed if the taps occur within a set period of time (this period is not configurable). If the second (or subsequent) taps do not occur within that period they are effectively ignored and the 'tap count' restarts.

## Using Xaml

A gesture recognizer can be added to a control in Xaml using attached properties. The syntax to add a `TapGestureRecognizer` to an image is shown below (in this case defining a *double tap* event):

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Tapped="OnTapGestureRecognizerTapped"
            NumberOfTapsRequired="2" />
    </Image.GestureRecognizers>
</Image>
```

The code for the event handler (in the sample) increments a counter and changes the image from color to black & white.

```
void OnTapGestureRecognizerTapped(object sender, EventArgs args)
{
    tapCount++;
    var imageSender = (Image)sender;
    // watch the monkey go from color to black&white!
    if (tapCount % 2 == 0) {
        imageSender.Source = "tapped.jpg";
    } else {
        imageSender.Source = "tapped_bw.jpg";
    }
}
```

## Using ICommand

Applications that use the Model-View-ViewModel (MVVM) pattern typically use `ICommand` rather than wiring up event handlers directly. The `TapGestureRecognizer` can easily support `ICommand` either by setting the binding in code:

```
var tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.SetBinding (TapGestureRecognizer.CommandProperty, "TapCommand");
image.GestureRecognizers.Add(tapGestureRecognizer);
```

or using Xaml:

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Command="{Binding TapCommand}"
            CommandParameter="Image1" />
    </Image.GestureRecognizers>
</Image>
```

The complete code for this view model can be found in the sample. The relevant `Command` implementation details are shown below:

```
public class TapViewModel : INotifyPropertyChanged
{
    int taps = 0;
    ICommand tapCommand;
    public TapViewModel () {
        // configure the TapCommand with a method
        tapCommand = new Command (OnTapped);
    }
    public ICommand TapCommand {
        get { return tapCommand; }
    }
    void OnTapped (object s) {
        taps++;
        Debug.WriteLine ("parameter: " + s);
    }
    //region INotifyPropertyChanged code omitted
}
```

## Related Links

- [TapGesture \(sample\)](#)
- [GestureRecognizer](#)
- [TapGestureRecognizer](#)

# Adding a pinch gesture recognizer

9/20/2018 • 3 minutes to read • [Edit Online](#)

The pinch gesture is used for performing interactive zoom and is implemented with the `PinchGestureRecognizer` class. A common scenario for the pinch gesture is to perform interactive zoom of an image at the pinch location. This is accomplished by scaling the content of the viewport, and is demonstrated in this article.

To make a user interface element zoomable with the pinch gesture, create a `PinchGestureRecognizer` instance, handle the `PinchUpdated` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the user interface element. The following code example shows a `PinchGestureRecognizer` attached to an `Image` element:

```
var pinchGesture = new PinchGestureRecognizer();
pinchGesture.PinchUpdated += (s, e) => {
    // Handle the pinch
};
image.GestureRecognizers.Add(pinchGesture);
```

This can also be achieved in XAML, as shown in the following code example:

```
<Image Source="waterfront.jpg">
<Image.GestureRecognizers>
    <PinchGestureRecognizer PinchUpdated="OnPinchUpdated" />
</Image.GestureRecognizers>
</Image>
```

The code for the `OnPinchUpdated` event handler is then added to the code-behind file:

```
void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
{
    // Handle the pinch
}
```

## Creating a PinchToZoom container

Handling the pinch gesture to perform a zoom operation requires some math to transform the user interface. This section contains a generalized helper class to perform the math, which can be used to interactively zoom any user interface element. The following code example shows the `PinchToZoomContainer` class:

```

public class PinchToZoomContainer : ContentView
{
    ...
    public PinchToZoomContainer ()
    {
        var pinchGesture = new PinchGestureRecognizer ();
        pinchGesture.PinchUpdated += OnPinchUpdated;
        GestureRecognizers.Add (pinchGesture);
    }

    void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
    {
        ...
    }
}

```

This class can be wrapped around a user interface element so that the pinch gesture will zoom the wrapped user interface element. The following XAML code example shows the `PinchToZoomContainer` wrapping an `Image` element:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:PinchGesture;assembly=PinchGesture"
    x:Class="PinchGesture.HomePage">
    <ContentPage.Content>
        <Grid Padding="20">
            <local:PinchToZoomContainer>
                <local:PinchToZoomContainer.Content>
                    <Image Source="waterfront.jpg" />
                </local:PinchToZoomContainer.Content>
            </local:PinchToZoomContainer>
        </Grid>
    </ContentPage.Content>
</ContentPage>

```

The following code example shows how the `PinchToZoomContainer` wraps an `Image` element in a C# page:

```

public class HomePageCS : ContentPage
{
    public HomePageCS ()
    {
        Content = new Grid {
            Padding = new Thickness (20),
            Children = {
                new PinchToZoomContainer {
                    Content = new Image { Source = ImageSource.FromFile ("waterfront.jpg") }
                }
            }
        };
    }
}

```

When the `Image` element receives a pinch gesture, the displayed image will be zoomed-in or out. The zoom is performed by the `PinchZoomContainer.OnPinchUpdated` method, which is shown in the following code example:

```

void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
{
    if (e.Status == GestureStatus.Started) {
        // Store the current scale factor applied to the wrapped user interface element,
        // and zero the components for the center point of the translate transform.
        startScale = Content.Scale;
        Content.AnchorX = 0;
        Content.AnchorY = 0;
    }
    if (e.Status == GestureStatus.Running) {
        // Calculate the scale factor to be applied.
        currentScale += (e.Scale - 1) * startScale;
        currentScale = Math.Max (1, currentScale);

        // The ScaleOrigin is in relative coordinates to the wrapped user interface element,
        // so get the X pixel coordinate.
        double renderedX = Content.X + xOffset;
        double deltaX = renderedX / Width;
        double deltaWidth = Width / (Content.Width * startScale);
        double originX = (e.ScaleOrigin.X - deltaX) * deltaWidth;

        // The ScaleOrigin is in relative coordinates to the wrapped user interface element,
        // so get the Y pixel coordinate.
        double renderedY = Content.Y + yOffset;
        double deltaY = renderedY / Height;
        double deltaHeight = Height / (Content.Height * startScale);
        double originY = (e.ScaleOrigin.Y - deltaY) * deltaHeight;

        // Calculate the transformed element pixel coordinates.
        double targetX = xOffset - (originX * Content.Width) * (currentScale - startScale);
        double targetY = yOffset - (originY * Content.Height) * (currentScale - startScale);

        // Apply translation based on the change in origin.
        Content.TranslationX = targetX.Clamp (-Content.Width * (currentScale - 1), 0);
        Content.TranslationY = targetY.Clamp (-Content.Height * (currentScale - 1), 0);

        // Apply scale factor.
        Content.Scale = currentScale;
    }
    if (e.Status == GestureStatus.Completed) {
        // Store the translation delta's of the wrapped user interface element.
        xOffset = Content.TranslationX;
        yOffset = Content.TranslationY;
    }
}

```

This method updates the zoom level of the wrapped user interface element based on the user's pinch gesture. This is achieved by using the values of the `Scale`, `ScaleOrigin` and `Status` properties of the `PinchGestureUpdatedEventArgs` instance to calculate the scale factor to be applied at the origin of the pinch gesture. The wrapped user element is then zoomed at the origin of the pinch gesture by setting its `TranslationX`, `TranslationY`, and `Scale` properties to the calculated values.

## Related Links

- [PinchGesture \(sample\)](#)
- [GestureRecognizer](#)
- [PinchGestureRecognizer](#)

# Adding a pan gesture recognizer

9/20/2018 • 3 minutes to read • [Edit Online](#)

The pan gesture is used for detecting the movement of fingers around the screen and applying that movement to content, and is implemented with the `PanGestureRecognizer` class. A common scenario for the pan gesture is to horizontally and vertically pan an image, so that all of the image content can be viewed when it's being displayed in a viewport smaller than the image dimensions. This is accomplished by moving the image within the viewport, and is demonstrated in this article.

To make a user interface element moveable with the pan gesture, create a `PanGestureRecognizer` instance, handle the `PanUpdated` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the user interface element. The following code example shows a `PanGestureRecognizer` attached to an `Image` element:

```
var panGesture = new PanGestureRecognizer();
panGesture.PanUpdated += (s, e) => {
    // Handle the pan
};
image.GestureRecognizers.Add(panGesture);
```

This can also be achieved in XAML, as shown in the following code example:

```
<Image Source="MonoMonkey.jpg">
<Image.GestureRecognizers>
    <PanGestureRecognizer PanUpdated="OnPanUpdated" />
</Image.GestureRecognizers>
</Image>
```

The code for the `OnPanUpdated` event handler is then added to the code-behind file:

```
void OnPanUpdated (object sender, PanUpdatedEventArgs e)
{
    // Handle the pan
}
```

## NOTE

Correct panning on Android requires the [Xamarin.Forms 2.1.0-pre1 NuGet package](#) at a minimum.

## Creating a pan container

This section contains a generalized helper class that performs freeform panning, which is typically suited to navigating within images or maps. Handling the pan gesture to perform this operation requires some math to transform the user interface. This math is used to pan only within the bounds of the wrapped user interface element. The following code example shows the `PanContainer` class:

```

public class PanContainer : ContentView
{
    double x, y;

    public PanContainer ()
    {
        // Set PanGestureRecognizer.TouchPoints to control the
        // number of touch points needed to pan
        var panGesture = new PanGestureRecognizer ();
        panGesture.PanUpdated += OnPanUpdated;
        GestureRecognizers.Add (panGesture);
    }

    void OnPanUpdated (object sender, PanUpdatedEventArgs e)
    {
        ...
    }
}

```

This class can be wrapped around a user interface element so that the gesture will pan the wrapped user interface element. The following XAML code example shows the `PanContainer` wrapping an `Image` element:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:PanGesture"
    x:Class="PanGesture.HomePage">
    <ContentPage.Content>
        <AbsoluteLayout>
            <local:PanContainer>
                <Image Source="MonoMonkey.jpg" WidthRequest="1024" HeightRequest="768" />
            </local:PanContainer>
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>

```

The following code example shows how the `PanContainer` wraps an `Image` element in a C# page:

```

public class HomePageCS : ContentPage
{
    public HomePageCS ()
    {
        Content = new AbsoluteLayout {
            Padding = new Thickness (20),
            Children = {
                new PanContainer {
                    Content = new Image {
                        Source = ImageSource.FromFile ("MonoMonkey.jpg"),
                        WidthRequest = 1024,
                        HeightRequest = 768
                    }
                }
            }
        };
    }
}

```

In both examples, the `WidthRequest` and `HeightRequest` properties are set to the width and height values of the image being displayed.

When the `Image` element receives a pan gesture, the displayed image will be panned. The pan is performed by the `PanContainer.OnPanUpdated` method, which is shown in the following code example:

```

void OnPanUpdated (object sender, PanUpdatedEventArgs e)
{
    switch (e.StatusType) {
        case GestureStatus.Running:
            // Translate and ensure we don't pan beyond the wrapped user interface element bounds.
            Content.TranslationX =
                Math.Max (Math.Min (0, x + e.TotalX), -Math.Abs (Content.Width - App.ScreenWidth));
            Content.TranslationY =
                Math.Max (Math.Min (0, y + e.TotalY), -Math.Abs (Content.Height - App.ScreenHeight));
            break;

        case GestureStatus.Completed:
            // Store the translation applied during the pan
            x = Content.TranslationX;
            y = Content.TranslationY;
            break;
    }
}

```

This method updates the viewable content of the wrapped user interface element, based on the user's pan gesture. This is achieved by using the values of the `TotalX` and `TotalY` properties of the `PanUpdatedEventArgs` instance to calculate the direction and distance of the pan. The `AppScreenWidth` and `AppScreenHeight` properties provide the height and width of the viewport, and are set to the screen width and screen height values of the device by the respective platform-specific projects. The wrapped user element is then panned by setting its `TranslationX` and `TranslationY` properties to the calculated values.

When panning content in an element that does not occupy the full screen, the height and width of the viewport can be obtained from the element's `Height` and `Width` properties.

#### **NOTE**

Displaying high-resolution images can greatly increase an app's memory footprint. Therefore, they should only be created when required and should be released as soon as the app no longer requires them. For more information, see [Optimize Image Resources](#).

## Related Links

- [PanGesture \(sample\)](#)
- [GestureRecognizer](#)
- [PanGestureRecognizer](#)

# Adding a swipe gesture recognizer

10/3/2018 • 5 minutes to read • [Edit Online](#)

A swipe gesture occurs when a finger is moved across the screen in a horizontal or vertical direction, and is often used to initiate navigation through content. The code examples in this article are taken from the [Swipe Gesture sample](#).

To make a `View` recognize a swipe gesture, create a `SwipeGestureRecognizer` instance, set the `Direction` property to a `SwipeDirection` enumeration value (`Left`, `Right`, `Up`, or `Down`), optionally set the `Threshold` property, handle the `Swiped` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the view.

The following code example shows a `SwipeGestureRecognizer` attached to a `BoxView`:

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>
```

Here is the equivalent C# code:

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
```

The `SwipeGestureRecognizer` class also includes a `Threshold` property, that can be optionally set to a `uint` value that represents the minimum swipe distance that must be achieved for a swipe to be recognized, in device-independent units. The default value of this property is 100, meaning that any swipes that are less than 100 device-independent units will be ignored.

## Recognizing the swipe direction

In the examples above, the `Direction` property is set to single a value from the `SwipeDirection` enumeration. However, it's also possible to set this property to multiple values from the `SwipeDirection` enumeration, so that the `Swiped` event is fired in response to a swipe in more than one direction. However, the constraint is that a single `SwipeGestureRecognizer` can only recognize swipes that occur on the same axis. Therefore, swipes that occur on the horizontal axis can be recognized by setting the `Direction` property to `Left` and `Right`:

```
<SwipeGestureRecognizer Direction="Left,Right" Swiped="OnSwiped"/>
```

Similarly, swipes that occur on the vertical axis can be recognized by setting the `Direction` property to `Up` and `Down`:

```
var swipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Up | SwipeDirection.Down };
```

Alternatively, a `SwipeGestureRecognizer` for each swipe direction can be created to recognize swipes in every direction:

```

<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Right" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Up" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Down" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>

```

Here is the equivalent C# code:

```

var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;
var rightSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Right };
rightSwipeGesture.Swiped += OnSwiped;
var upSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Up };
upSwipeGesture.Swiped += OnSwiped;
var downSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Down };
downSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
boxView.GestureRecognizers.Add(rightSwipeGesture);
boxView.GestureRecognizers.Add(upSwipeGesture);
boxView.GestureRecognizers.Add(downSwipeGesture);

```

#### NOTE

In the above examples, the same event handler responds to the `Swiped` event firing. However, each `SwipeGestureRecognizer` instance can use a different event handler if required.

## Responding to the swipe

An event handler for the `Swiped` event is shown in the following example:

```

void OnSwiped(object sender, SwipedEventArgs e)
{
    switch (e.Direction)
    {
        case SwipeDirection.Left:
            // Handle the swipe
            break;
        case SwipeDirection.Right:
            // Handle the swipe
            break;
        case SwipeDirection.Up:
            // Handle the swipe
            break;
        case SwipeDirection.Down:
            // Handle the swipe
            break;
    }
}

```

The `SwipedEventArgs` can be examined to determine the direction of the swipe, with custom logic responding to the swipe as required. The direction of the swipe can be obtained from the `Direction` property of the event arguments, which will be set to one of the values of the `SwipeDirection` enumeration. In addition, the event arguments also have a `Parameter` property that will be set to the value of the `CommandParameter` property, if

defined.

## Using commands

The `SwipeGestureRecognizer` class also includes `Command` and `CommandParameter` properties. These properties are typically used in applications that use the Model-View-ViewModel (MVVM) pattern. The `Command` property defines the `ICommand` to be invoked when a swipe gesture is recognized, with the `CommandParameter` property defining an object to be passed to the `ICommand`. The following code example shows how to bind the `Command` property to an `ICommand` defined in the view model whose instance is set as the page `BindingContext`:

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left, CommandParameter = "Left" };
leftSwipeGesture.SetBinding(SwipeGestureRecognizer.CommandProperty, "SwipeCommand");
boxView.GestureRecognizers.Add(leftSwipeGesture);
```

The equivalent XAML code is:

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Command="{Binding SwipeCommand}" CommandParameter="Left" />
    </BoxView.GestureRecognizers>
</BoxView>
```

`SwipeCommand` is a property of type `ICommand` defined in the view model instance that is set as the page `BindingContext`. When a swipe gesture is recognized, the `Execute` method of the `SwipeCommand` object will be executed. The argument to the `Execute` method is the value of the `CommandParameter` property. For more information about commands, see [The Command Interface](#).

## Creating a swipe container

The `SwipeContainer` class, which is shown in the following code example, is a generalized swipe recognition class that can be wrapped around a `View` to perform swipe gesture recognition:

```
public class SwipeContainer : ContentView
{
    public event EventHandler<SwipedEventArgs> Swipe;

    public SwipeContainer()
    {
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Left));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Right));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Up));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Down));
    }

    SwipeGestureRecognizer GetSwipeGestureRecognizer(SwipeDirection direction)
    {
        var swipe = new SwipeGestureRecognizer { Direction = direction };
        swipe.Swiped += (sender, e) => Swipe?.Invoke(this, e);
        return swipe;
    }
}
```

The `SwipeContainer` class creates `SwipeGestureRecognizer` objects for all four swipe directions, and attaches `Swipe` event handlers. These event handlers invoke the `Swipe` event defined by the `SwipeContainer`.

The following XAML code example shows the `SwipeContainer` class wrapping a `BoxView`:

```
<ContentPage ...>
    <StackLayout>
        <local:SwipeContainer Swipe="OnSwiped" ...>
            <BoxView Color="Teal" ... />
        </local:SwipeContainer>
    </StackLayout>
</ContentPage>
```

The following code example shows how the `SwipeContainer` wraps a `BoxView` in a C# page:

```
public class SwipeContainerPageCS : ContentPage
{
    public SwipeContainerPageCS()
    {
        var boxView = new BoxView { Color = Color.Teal, ... };
        var swipeContainer = new SwipeContainer { Content = boxView, ... };
        swipeContainer.Swipe += (sender, e) =>
        {
            // Handle the swipe
        };

        Content = new StackLayout
        {
            Children = { swipeContainer }
        };
    }
}
```

When the `BoxView` receives a swipe gesture, the `Swiped` event in the `SwipeGestureRecognizer` is fired. This is handled by the `SwipeContainer` class, which fires its own `Swipe` event. This `Swipe` event is handled on the page. The `SwipedEventArgs` can then be examined to determine the direction of the swipe, with custom logic responding to the swipe as required.

## Related links

- [Swipe Gesture \(sample\)](#)
- [GestureRecognizer](#)
- [SwipeGestureRecognizer](#)

# Xamarin.Forms Localization

11/8/2018 • 2 minutes to read • [Edit Online](#)

*The built-in .NET localization framework can be used to build cross-platform multilingual applications with Xamarin.Forms.*

## String and Image Localization

The built-in mechanism for localizing .NET applications uses [RESX files](#) and the classes in the `System.Resources` and `System.Globalization` namespaces. The RESX files containing translated strings are embedded in the Xamarin.Forms assembly, along with a compiler-generated class that provides strongly-typed access to the translations. The translated text can then be retrieved in code.

## Right-to-Left Localization

Flow direction is the direction in which the UI elements on the page are scanned by the eye. Right-to-left localization adds support for right-to-left flow direction to Xamarin.Forms applications.

# Localization

10/17/2018 • 24 minutes to read • [Edit Online](#)

Xamarin.Forms apps can be localized using .NET resources files.

## Overview

The built-in mechanism for localizing .NET applications uses [RESX files](#) and the classes in the `System.Resources` and `System.Globalization` namespaces. The RESX files containing translated strings are embedded in the Xamarin.Forms assembly, along with a compiler-generated class that provides strongly-typed access to the translations. The translated text can then be retrieved in code.

### Sample Code

There are two samples associated with this document:

- [UsingResxLocalization](#) is a very simple demonstration of the concepts explained. The code snippets shown below are all from this sample.
- [TodoLocalized](#) is a basic working app that uses these localization techniques.

#### Shared Projects are not recommended

The TodoLocalized sample includes a [Shared Project demo](#) however due to limitations of the build system the resource files do not get a `.designer.cs` file generated which breaks the ability to access translated strings strongly-typed in code.

The remainder of this document relates to projects using the Xamarin.Forms .NET Standard library template.

## Globalizing Xamarin.Forms Code

**Globalizing** an application is the process of making it "world ready." This means writing code that is capable of displaying different languages.

One of the key parts of globalizing an application is building the user-interface so that there is no *hard-coded* text. Instead, anything displayed to the user should be retrieved from a set of strings that have been translated into their chosen language.

In this document we'll examine how to use RESX files to store those strings and retrieve them for display depending on the user's preference.

The samples target English, French, Spanish, German, Chinese, Japanese, Russian, and Brazilian Portuguese languages. Applications can be translated into as few or as many languages as required.

#### NOTE

On the Universal Windows Platform, RESW files should be used for push notification localization, rather than RESX files. For more information, see [UWP Localization](#).

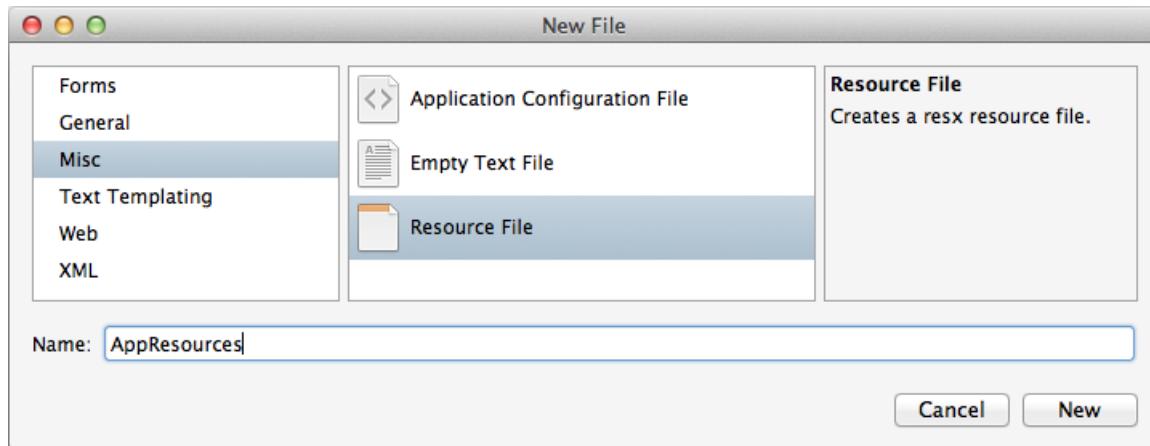
### Adding Resources

The first step in globalizing a Xamarin.Forms .NET Standard library application is adding the RESX resource files that will be used to store all the text used in the app. We need to add a RESX file that contains the default text, and then add additional RESX files for each language we wish to support.

#### Base Language Resource

The base resources (RESX) file will contain the default language strings (the samples assume English is the default language). Add the file to the Xamarin.Forms common code project by right-clicking on the project and choosing **Add > New File....**

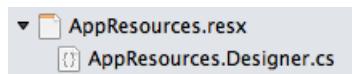
Choose a meaningful name such as **AppResources** and press **OK**.



Two files will be added to the project:

- **AppResources.resx** file where translatable strings are stored in an XML format.
- **AppResources.designer.cs** file that declares a partial class to contain references to all the elements created in the RESX XML file.

The solution tree will show the files as related. The RESX file *should* be edited to add new translatable strings; the **.designer.cs** file should *not* be edited.

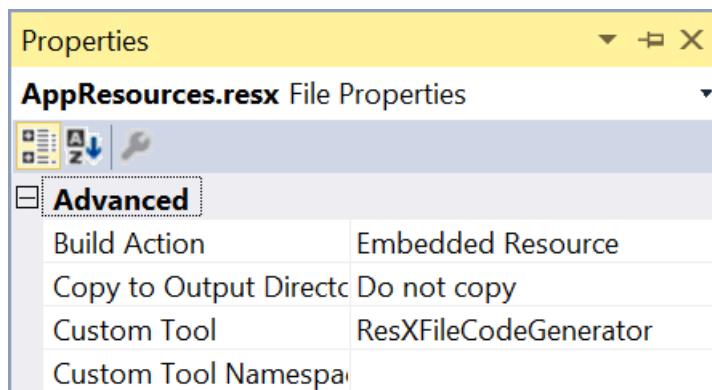


#### String Visibility

By default when strongly-typed references to strings are generated, they will be `internal` to the assembly. This is because the default build tool for RESX files generates the **.designer.cs** file with `internal` properties.

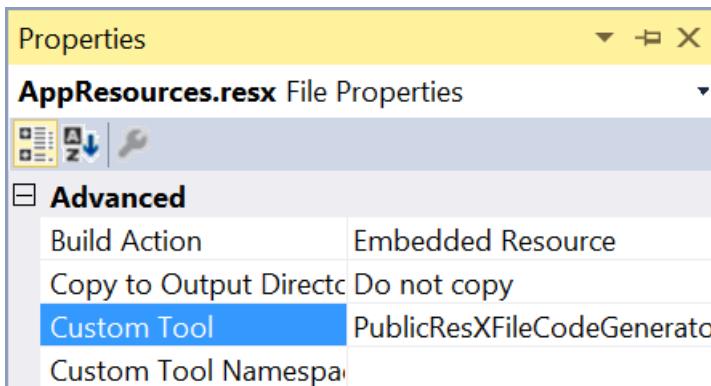
Select the **AppResources.resx** file and show the **Properties** pad to see where this build tool is configured. The screenshot below shows the **Custom Tool: ResXFileCodeGenerator**.

- [Visual Studio](#)
- [Visual Studio for Mac](#)



To make the strongly-typed string properties `public`, you must manually change the configuration to **Custom Tool: PublicResXFileCodeGenerator**, as shown in the screenshot below:

- [Visual Studio](#)
- [Visual Studio for Mac](#)



This change is optional, and is only required if you wish to reference localized strings across different assemblies (for example, if you put the RESX files in a different assembly to your code). The sample for this topic leaves the strings `internal` because they are defined in the same `Xamarin.Forms .NET Standard library` assembly where they are used.

You only need to set the custom tool on the base RESX file as shown above; you do not need to set *any* build tool on the language-specific RESX files discussed in the following sections.

#### Editing the RESX file

Unfortunately there is no built-in RESX editor in Visual Studio for Mac. Adding new translatable strings requires the addition of a new XML `data` element for each string. Each `data` element can contain the following:

- `name` attribute (required) is the key for this translatable string. It must be a valid C# property name - so no spaces or special characters are allowed.
- `value` element (required), which is the actual string that is displayed in the application.
- `comment` element (optional) can contain instructions for the translator that explains how this string is used.
- `xml:space` attribute (optional) to control how spacing in the string is preserved.

Some example `data` elements are shown here:

```
<data name="NotesLabel" xml:space="preserve">
    <value>Notes:</value>
    <comment>label for input field</comment>
</data>
<data name="NotesPlaceholder" xml:space="preserve">
    <value>eg. buy milk</value>
    <comment>example input for notes field</comment>
</data>
<data name="AddButton" xml:space="preserve">
    <value>Add new item</value>
</data>
```

As the application is written, every piece of text displayed to the user should be added to the base RESX resources file in a new `data` element. It is recommended that you include `comment`s as much as possible to ensure a high-quality translation.

#### NOTE

Visual Studio (including the free Community edition) contains a basic RESX editor. If you have access to a Windows computer, that can be a convenient way to add and edit strings in RESX files.

#### Language-Specific Resources

Typically, the actual translation of the default text strings won't happen until large chunks of the application have been written (in which case the default RESX file will contain lots of strings). It is still a good idea to add the language-specific resources early in the development cycle, optionally populating with machine-translated text to

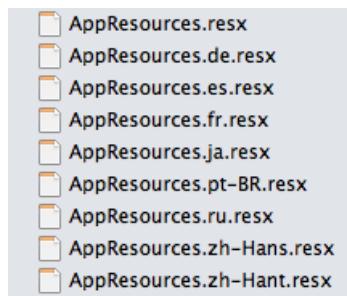
help test the localization code.

One additional RESX file is added for each language we wish to support. Language-specific resource files must follow a specific naming convention: use the same filename as the base resources file (eg. **AppResources**) followed by a period (.) and then the language code. Simple examples include:

- **AppResources.fr.resx** - French language translations.
- **AppResources.es.resx** - Spanish language translations.
- **AppResources.de.resx** - German language translations.
- **AppResources.ja.resx** - Japanese language translations.
- **AppResources.zh-Hans.resx** - Chinese (Simplified) language translations.
- **AppResources.zh-Hant.resx** - Chinese (Traditional) language translations.
- **AppResources.pt.resx** - Portuguese language translations.
- **AppResources.pt-BR.resx** - Brazilian Portuguese language translations.

The general pattern is to use two-letter language codes, but there are some examples (such as Chinese) where a different format is used, and other examples (such as Brazilian Portuguese) where a four character locale identifier is required.

These language-specific resources files *do not* require a **.designer.cs** partial class so they can be added as regular XML files, with the **Build Action: EmbeddedResource** set. This screenshot shows a solution containing language-specific resource files:



As an application is developed and the base RESX file has text added, you should send it out to translators who will translate each `data` element and return a language-specific resource file (using the naming convention shown) to be included in the app. Some 'machine translated' examples are shown below:

### AppResources.es.resx (Spanish)

```
<data name="AddButton" xml:space="preserve">
    <value>Escribir un articulo</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

### AppResources.ja.resx (Japanese)

```
<data name="AddButton" xml:space="preserve">
    <value>新しい項目を追加</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

### AppResources.pt-BR.resx (Brazilian Portuguese)

```
<data name="AddButton" xml:space="preserve">
    <value>adicionar novo item</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

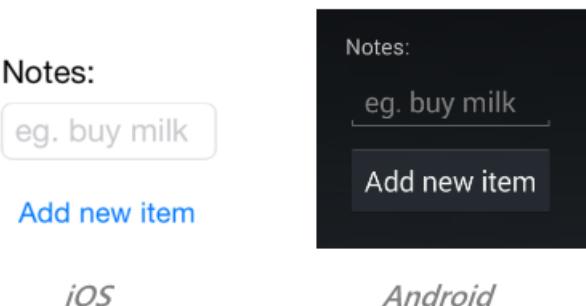
Only the `value` element needs to be updated by the translator - the `comment` is not intended to be translated. Remember: when editing XML files to escape reserved characters like `<`, `>`, `&` with `&lt;`, `&gt;`, and `&amp;` if they appear in the `value` or `comment`.

## Using Resources in Code

Strings in the RESX resource files will be available to use in your user interface code using the `AppResources` class. The `name` assigned to each string in the RESX file becomes a property on that class which can be referenced in Xamarin.Forms code as shown below:

```
var myLabel = new Label ();
var myEntry = new Entry ();
var myButton = new Button ();
// populate UI with translated text values from resources
myLabel.Text = AppResources.NotesLabel;
myEntry.Placeholder = AppResources.NotesPlaceholder;
myButton.Text = AppResources.AddButton;
```

The user interface on iOS, Android, and the Universal Windows Platform (UWP) renders as you'd expect, except now it's possible to translate the app into multiple languages because the text is being loaded from a resource rather than being hard-coded. Here is a screenshot showing the UI on each platform prior to translation:



iOS

Android

## Troubleshooting

### Testing a Specific Language

It can be tricky to switch the simulator or a device to different languages, particularly during development when you want to test different cultures quickly.

You can force a specific language to be loaded by setting the `culture` as shown in this code snippet:

```
// force a specific culture, useful for quick testing
AppResources.Culture = new CultureInfo("fr-FR");
```

This approach – setting the culture directly on the `AppResources` class – can also be used to implement a language-selector inside your app (rather than using the device's locale).

### Loading Embedded Resources

The following code snippet is useful when trying to debug issues with embedded resources (such as RESX files). Add this code to your app (early in the app lifecycle) and it will list all the resources embedded in the assembly, showing the full resource identifier:

```

using System.Reflection;
// ...
// NOTE: use for debugging, not in released app code!
var assembly = typeof(EmbeddedImages).GetTypeInfo().Assembly; // "EmbeddedImages" should be a class in your
app
foreach (var res in assembly.GetManifestResourceNames())
{
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}

```

In the **AppResources.Designer.cs** file (around *line 33*), the full *resource manager name* is specified (eg. "UsingResxLocalization.Resx.AppResources" ) similar to the code below:

```

System.Resources.ResourceManager temp =
    new System.Resources.ResourceManager(
        "UsingResxLocalization.Resx.AppResources",
        typeof(AppResources).GetTypeInfo().Assembly);

```

Check the **Application Output** for the results of the debug code shown above, to confirm the correct resources are listed (ie. "UsingResxLocalization.Resx.AppResources" ).

If not, the **AppResources** class will be unable to load its resources. Check the following to resolve issues where the resources cannot be found:

- The default namespace for the project matches the root namespace in the **AppResources.Designer.cs** file.
- If the **AppResources.resx** file is located in a subdirectory, the subdirectory name should be part of the namespace *and* part of the resource identifier.
- The **AppResources.resx** file has **Build Action: EmbeddedResource**.
- The **Project Options > Source Code > .NET Naming Policies > Use Visual Studio-style resources names** is ticked. You can untick this if you prefer, however the namespaces used when referencing your RESX resources will need to updated throughout the app.

#### **Doesn't work in DEBUG mode (Android only)**

If the translated strings are working in your RELEASE Android builds but not while debugging, right-click on the **Android Project** and select **Options > Build > Android Build** and ensure that the **Fast assembly deployment** is NOT ticked. This option causes problems with loading resources and should not be used if you are testing localized apps.

### **Displaying the Correct Language**

So far we've examined how to write code so that translations can be provided, but not how to actually make them appear. Xamarin.Forms code can take advantage of .NET's resources to load the correct language translations, but we need to query the operating system on each platform to determine which language the user has selected.

Because some platform-specific code is required to obtain the user's language preference, use a [dependency service](#) to expose that information in the Xamarin.Forms app and implement it for each platform.

First, define an interface to expose the user's preferred culture, similar to the code below:

```

public interface ILocalize
{
    CultureInfo GetCurrentCultureInfo ();
    void SetLocale (CultureInfo ci);
}

```

Second, use the [DependencyService](#) in the Xamarin.Forms **App** class to call the interface and set our RESX resources culture to the correct value. Notice that we don't need to manually set this value for the Universal

Windows Platform, since the resources framework automatically recognizes the selected language on those platforms.

```
if (Device.RuntimePlatform == Device.iOS || Device.RuntimePlatform == Device.Android)
{
    var ci = DependencyService.Get<ILocalize>().GetCurrentCultureInfo();
    Resx.AppResources.Culture = ci; // set the RESX for resource localization
    DependencyService.Get<ILocalize>().SetLocale(ci); // set the Thread for locale-aware methods
}
```

The resource `Culture` needs to be set when the application first loads so that the correct language strings are used. You may optionally update this value according to platform-specific events that may be raised on iOS or Android if the user updates their language preferences while the app is running.

The implementations for the `ILocalize` interface are shown in the [Platform-specific Code](#) section below. These implementations take advantage of this `PlatformCulture` helper class:

```
public class PlatformCulture
{
    public PlatformCulture (string platformCultureString)
    {
        if (String.IsNullOrEmpty(platformCultureString))
        {
            throw new ArgumentException("Expected culture identifier", "platformCultureString"); // in C# 6
use nameof(platformCultureString)
        }
        PlatformString = platformCultureString.Replace("_", "-"); // .NET expects dash, not underscore
        var dashIndex = PlatformString.IndexOf("-", StringComparison.Ordinal);
        if (dashIndex > 0)
        {
            var parts = PlatformString.Split('-');
            LanguageCode = parts[0];
            LocaleCode = parts[1];
        }
        else
        {
            LanguageCode = PlatformString;
            LocaleCode = "";
        }
    }
    public string PlatformString { get; private set; }
    public string LanguageCode { get; private set; }
    public string LocaleCode { get; private set; }
    public override string ToString()
    {
        return PlatformString;
    }
}
```

## Platform-Specific Code

The code to detect which language to display must be platform-specific because iOS, Android, and UWP all expose this information in slightly different ways. The code for the `ILocalize` dependency service is provided below for each platform, along with additional platform-specific requirements to ensure localized text is rendered correctly.

The platform-specific code must also handle cases where the operating system allows the user to configure a locale identifier that is not supported by .NET's `CultureInfo` class. In these cases custom code must be written to detect unsupported locales and substitute the best .NET-compatible locale.

### iOS Application Project

iOS users select their preferred language separately from date and time formatting culture. To load the correct

resources to localize a Xamarin.Forms app we just need to query the `NSLocale.PreferredLanguages` array for the first element.

The following implementation of the `ILocalize` dependency service should be placed in the iOS application project. Because iOS uses underscores instead of dashes (which is the .NET standard representation) the code replaces the underscore before instantiating the `CultureInfo` class:

```
[assembly:Dependency(typeof(UsingResxLocalization.iOS.Localize))]

namespace UsingResxLocalization.iOS
{
    public class Localize : UsingResxLocalization.ILocalize
    {
        public void SetLocale (CultureInfo ci)
        {
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
        public CultureInfo GetCurrentCultureInfo ()
        {
            var netLanguage = "en";
            if (NSLocale.PreferredLanguages.Length > 0)
            {
                var pref = NSLocale.PreferredLanguages [0];
                netLanguage = iOSToDotnetLanguage(pref);
            }
            // this gets called a lot - try/catch can be expensive so consider caching or something
            System.Globalization.CultureInfo ci = null;
            try
            {
                ci = new System.Globalization.CultureInfo(netLanguage);
            }
            catch (CultureNotFoundException e1)
            {
                // iOS locale not valid .NET culture (eg. "en-ES" : English in Spain)
                // fallback to first characters, in this case "en"
                try
                {
                    var fallback = ToDotnetFallbackLanguage(new PlatformCulture(netLanguage));
                    ci = new System.Globalization.CultureInfo(fallback);
                }
                catch (CultureNotFoundException e2)
                {
                    // iOS language not valid .NET culture, falling back to English
                    ci = new System.Globalization.CultureInfo("en");
                }
            }
            return ci;
        }
        string iOSToDotnetLanguage(string iOSLanguage)
        {
            var netLanguage = iOSLanguage;
            //certain languages need to be converted to CultureInfo equivalent
            switch (iOSLanguage)
            {
                case "ms-MY": // "Malaysian (Malaysia)" not supported .NET culture
                case "ms-SG": // "Malaysian (Singapore)" not supported .NET culture
                    netLanguage = "ms"; // closest supported
                    break;
                case "gsw-CH": // "Schwiizertüütsch (Swiss German)" not supported .NET culture
                    netLanguage = "de-CH"; // closest supported
                    break;
                // add more application-specific cases here (if required)
                // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
    }
}
```

```

        ,
        string ToDotnetFallbackLanguage (PlatformCulture platCulture)
        {
            var netLanguage = platCulture.LanguageCode; // use the first part of the identifier (two chars,
            usually);
            switch (platCulture.LanguageCode)
            {
                case "pt":
                    netLanguage = "pt-PT"; // fallback to Portuguese (Portugal)
                    break;
                case "gsw":
                    netLanguage = "de-CH"; // equivalent to German (Switzerland) for this app
                    break;
                // add more application-specific cases here (if required)
                // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
    }
}

```

#### NOTE

The `try/catch` blocks in the `GetCurrentCultureInfo` method mimic the fallback behavior typically used with locale specifiers – if the exact match is not found, look for a close match based just on the language (first block of characters in the locale).

In the case of Xamarin.Forms, some locales are valid in iOS but do not correspond to a valid `cultureInfo` in .NET, and the code above attempts to handle this.

For example, the iOS **Settings > General Language & Region** screen allows you to set your phone **Language** to **English** but the **Region** to **Spain**, which results in a locale string of `"en-ES"`. When the `cultureInfo` creation fails, the code falls back to using just the first two letters to select the display language.

Developers should modify the `iOSToDotnetLanguage` and `ToDotnetFallbackLanguage` methods to handle specific cases required for their supported languages.

There are some system-defined user-interface elements that are automatically translated by iOS, such as the **Done** button on the `Picker` control. To force iOS to translate these elements we need to indicate which languages we support in the **Info.plist** file. You can add these values via **Info.plist > Source** as shown here:

<b>▼ Localizations</b>	<b>Array</b>	<b>(9 items)</b>
	String	de
	String	es
	String	fr
	String	ja
	String	pt
	String	pt-PT
	String	ru
	String	zh-Hans
	String	zh-Hant
<b>Add new entry</b>		
<b>Localization native deve</b>	<b>String</b>	<b>en</b>

Alternatively, open the **Info.plist** file in an XML editor and edit the values directly:

```

<key>CFBundleLocalizations</key>
<array>
    <string>de</string>
    <string>es</string>
    <string>fr</string>
    <string>ja</string>
    <string>pt</string> <!-- Brazil -->
    <string>pt-PT</string> <!-- Portugal -->
    <string>ru</string>
    <string>zh-Hans</string>
    <string>zh-Hant</string>
</array>
<key>CFBundleDevelopmentRegion</key>
<string>en</string>

```

Once you've implemented the dependency service and updated **Info.plist**, the iOS app will be able to display localized text.

#### NOTE

Note that Apple treats Portuguese slightly differently than you'd expect. From [their docs](#): "use *pt* as the language ID for Portuguese as it is used in Brazil and *pt-PT* as the language ID for Portuguese as it is used in Portugal". This means when Portuguese language is chosen in a non-standard locale, the fallback language will be Brazilian Portuguese on iOS, unless code is written to change this behavior (such as the `ToDotnetFallbackLanguage` above).

For more information about iOS Localization, see [iOS Localization](#).

#### Android Application Project

Android exposes the currently selected locale via `Java.Util.Locale.Default`, and also uses an underscore separator instead of a dash (which is replaced by the following code). Add this dependency service implementation to the Android application project:

```

[assembly:Dependency(typeof(UsingResxLocalization.Android.Localize))]

namespace UsingResxLocalization.Android
{
    public class Localize : UsingResxLocalization.ILocalize
    {
        public void SetLocale(CultureInfo ci)
        {
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
        public CultureInfo GetCurrentCultureInfo()
        {
            var netLanguage = "en";
            var androidLocale = Java.Util.Locale.Default;
            netLanguage = AndroidToDotnetLanguage(androidLocale.ToString().Replace("_", "-"));
            // this gets called a lot - try/catch can be expensive so consider caching or something
            System.Globalization.CultureInfo ci = null;
            try
            {
                ci = new System.Globalization.CultureInfo(netLanguage);
            }
            catch (CultureNotFoundException e1)
            {
                // iOS locale not valid .NET culture (eg. "en-ES" : English in Spain)
                // fallback to first characters, in this case "en"
                try
                {
                    var fallback = ToDotnetFallbackLanguage(new PlatformCulture(netLanguage));
                    ci = new System.Globalization.CultureInfo(fallback);
                }
                catch { }
            }
        }
    }
}

```

```

        }
        catch (CultureNotFoundException e2)
        {
            // iOS language not valid .NET culture, falling back to English
            ci = new System.Globalization.CultureInfo("en");
        }
    }
    return ci;
}
string AndroidToDotnetLanguage(string androidLanguage)
{
    var netLanguage = androidLanguage;
    //certain languages need to be converted to CultureInfo equivalent
    switch (androidLanguage)
    {
        case "ms-BN":    // "Malaysian (Brunei)" not supported .NET culture
        case "ms-MY":    // "Malaysian (Malaysia)" not supported .NET culture
        case "ms-SG":    // "Malaysian (Singapore)" not supported .NET culture
            netLanguage = "ms"; // closest supported
            break;
        case "in-ID":   // "Indonesian (Indonesia)" has different code in .NET
            netLanguage = "id-ID"; // correct code for .NET
            break;
        case "gsw-CH":  // "Schwiizertüütsch (Swiss German)" not supported .NET culture
            netLanguage = "de-CH"; // closest supported
            break;
            // add more application-specific cases here (if required)
            // ONLY use cultures that have been tested and known to work
    }
    return netLanguage;
}
string ToDotnetFallbackLanguage(PlatformCulture platCulture)
{
    var netLanguage = platCulture.LanguageCode; // use the first part of the identifier (two chars,
usually);
    switch (platCulture.LanguageCode)
    {
        case "gsw":
            netLanguage = "de-CH"; // equivalent to German (Switzerland) for this app
            break;
            // add more application-specific cases here (if required)
            // ONLY use cultures that have been tested and known to work
    }
    return netLanguage;
}
}
}
}

```

## NOTE

The `try/catch` blocks in the `GetCurrentCultureInfo` method mimic the fallback behavior typically used with locale specifiers – if the exact match is not found, look for a close match based just on the language (first block of characters in the locale).

In the case of Xamarin.Forms, some locales are valid in Android but do not correspond to a valid `CultureInfo` in .NET, and the code above attempts to handle this.

Developers should modify the `iOSToDotnetLanguage` and `ToDotnetFallbackLanguage` methods to handle specific cases required for their supported languages.

Once this code has been added to the Android application project, it will be able to automatically display translated strings.

## NOTE

□ **WARNING:** If the translated strings are working in your RELEASE Android builds but not while debugging, right-click on the **Android Project** and select **Options > Build > Android Build** and ensure that the **Fast assembly deployment** is NOT ticked. This option causes problems with loading resources and should not be used if you are testing localized apps.

For more information about Android localization, see [Android Localization](#).

## Universal Windows Platform

Universal Windows Platform (UWP) projects do not require the dependency service. Instead, this platform automatically sets the resource's culture correctly.

### AssemblyInfo.cs

Expand the Properties node in the .NET Standard library project and double-click on the **AssemblyInfo.cs** file. Add the following line to the file to set the neutral resources assembly language to English:

```
[assembly: NeutralResourcesLanguage("en")]
```

This informs the resource manager of the app's default culture, therefore ensuring that the strings defined in the language neutral RESX file (**AppResources.resx**) will be displayed when the app is running in one of the English locales.

## Example

After updating the platform-specific projects as shown above and recompiling the app with translated RESX files, updated translations will be available in each app. Here is a screenshot from the sample code translated into Simplified Chinese:



For more information about UWP localization, see [UWP Localization](#).

## Localizing XAML

When building a Xamarin.Forms user interface in XAML the markup would look similar to this, with strings embedded directly in the XML:

```
<Label Text="Notes:" />
<Entry Placeholder="eg. buy milk" />
<Button Text="Add to list" />
```

Ideally, we could translate user interface controls directly in the XAML, which we can do by creating a *markup extension*. The code for a markup extension that exposes the RESX resources to XAML is shown below. This class should be added to the Xamarin.Forms common code (along with the XAML pages):

```

using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

namespace UsingResxLocalization
{
    // You exclude the 'Extension' suffix when using in XAML
    [ContentProperty("Text")]
    public class TranslateExtension : IMarkupExtension
    {
        readonly CultureInfo ci = null;
        const string ResourceId = "UsingResxLocalization.Resx.AppResources";

        static readonly Lazy<ResourceManager> ResMgr = new Lazy<ResourceManager>(
            () => new ResourceManager(ResourceId,
                IntrospectionExtensions.GetTypeInfo(typeof(TranslateExtension)).Assembly));

        public string Text { get; set; }

        public TranslateExtension()
        {
            if (Device.RuntimePlatform == Device.iOS || Device.RuntimePlatform == Device.Android)
            {
                ci = DependencyService.Get<ILocalize>().GetCurrentCultureInfo();
            }
        }

        public object ProvideValue(IServiceProvider serviceProvider)
        {
            if (Text == null)
                return string.Empty;

            var translation = ResMgr.Value.GetString(Text, ci);
            if (translation == null)
            {
                #if DEBUG
                    throw new ArgumentException(
                        string.Format("Key '{0}' was not found in resources '{1}' for culture '{2}'.", Text,
                        ResourceId, ci.Name),
                        "Text");
                #else
                    translation = Text; // HACK: returns the key, which GETS DISPLAYED TO THE USER
                #endif
            }
            return translation;
        }
    }
}

```

The following bullets explain the important elements in the code above:

- The class is named `TranslateExtension`, but by convention we can refer to it as **Translate** in our markup.
- The class implements `IMarkupExtension`, which is required by Xamarin.Forms for it to work.
- `"UsingResxLocalization.Resx.AppResources"` is the resource identifier for our RESX resources. It is comprised of our default namespace, the folder where the resource files are located and the default RESX filename.
- The `ResourceManager` class is created using `IntrospectionExtensions.GetTypeInfo(typeof(TranslateExtension)).Assembly` to determine the current assembly to load resources from, and cached in the static `ResMgr` field. It's created as a `Lazy` type so that its creation is deferred until it's first used in the `ProvideValue` method.
- `ci` uses the dependency service to get the user's chosen language from the native operating system.

- `GetString` is the method that retrieves the actual translated string from the resources files. On the Universal Windows Platform, `ci` will be null because the `ILocalize` interface isn't implemented on those platforms. This is equivalent to calling the `GetString` method with only the first parameter. Instead, the resources framework will automatically recognize the locale and will retrieve the translated string from the appropriate RESX file.
- Error handling has been included to help debug missing resources by throwing an exception (in `DEBUG` mode only).

The following XAML snippet shows how to use the markup extension. There are two steps to make it work:

1. Declare the custom `xmlns:i18n` namespace in the root node. The `namespace` and `assembly` must match the project settings exactly - in this example they are identical but could be different in your project.
2. Use `{Binding}` syntax on attributes that would normally contain text to call the `Translate` markup extension. The resource key is the only required parameter.

```
<?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="UsingResxLocalization.FirstPageXaml"
    xmlns:i18n="clr-namespace:UsingResxLocalization;assembly=UsingResxLocalization">
    <StackLayout Padding="0, 20, 0, 0">
        <Label Text="{i18n:Translate NotesLabel}" />
        <Entry Placeholder="{i18n:Translate NotesPlaceholder}" />
        <Button Text="{i18n:Translate AddButton}" />
    </StackLayout>
</ContentPage>
```

The following more verbose syntax is also valid for the markup extension:

```
<Button Text="{i18n:TranslateExtension Text=AddButton}" />
```

## Localizing Platform-Specific Elements

Although we can handle the translation of the user interface in Xamarin.Forms code, there are some elements that must be done in each platform-specific project. This section will cover how to localize:

- Application Name
- Images

The sample project includes a localized image called **flag.png**, which is referenced in C# as follows:

```
var flag = new Image();
switch (Device.RuntimePlatform)
{
    case Device.iOS:
    case Device.Android:
        flag.Source = ImageSource.FromFile("flag.png");
        break;
    case Device.UWP:
        flag.Source = ImageSource.FromFile("Assets/Images/flag.png");
        break;
}
```

The flag image is also referenced in the XAML like this:

```

<Image>
  <Image.Source>
    <OnPlatform x:TypeArguments="ImageSource">
      <On Platform="iOS, Android" Value="flag.png" />
      <On Platform="UWP" Value="Assets/Images/flag.png" />
    </OnPlatform>
  </Image.Source>
</Image>

```

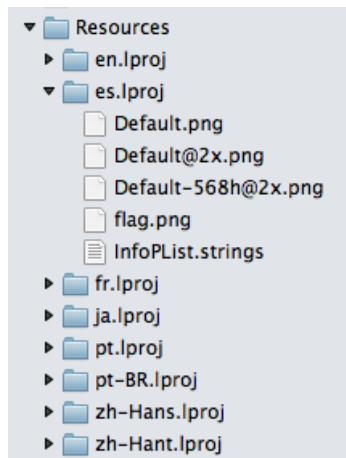
All platforms will automatically resolve image references like these to localized versions of the images, as long as the project structures explained below are implemented.

## iOS Application Project

iOS uses a naming standard called Localization Projects or **.lproj** directories to contain image and string resources. These directories can contain localized versions of images used in the app, and also the **InfoPlist.strings** file that can be used to localize the app name. For more information about iOS Localization, see [iOS Localization](#).

### Images

This screenshot shows the iOS sample app with language-specific **.lproj** directories. The Spanish directory called **es.lproj**, contains localized versions of the default image, as well as **flag.png**:



Each language directory contains a copy of **flag.png**, localized for that language. If no image is supplied, the operating system will default to the image in the default language directory. For full Retina support, you should supply **@2x** and **@3x** copies of each image.

### App Name

The contents of the **InfoPlist.strings** is just a single key-value to configure the app name:

```
"CFBundleDisplayName" = "ResxEspañol";
```

When the application is run, the app name and the image are both localized:

**Notas:**

por ejemplo . comprar leche



Agregar nuevo elemento

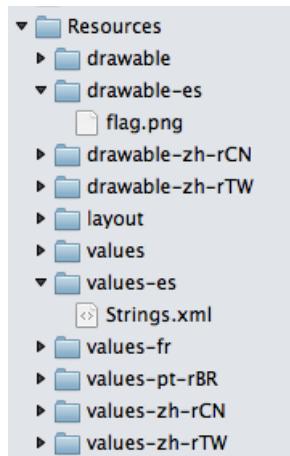


## Android Application Project

Android follows a different scheme for storing localized images using different **drawable** and **strings** directories with a language code suffix. When a four-letter locale code is required (such as zh-TW or pt-BR), note that Android requires an additional **r** following the dash/preceding the locale code (eg. zh-rTW or pt-rBR). For more information about Android localization, see [Android Localization](#).

### Images

This screenshot shows the Android sample with some localized drawables and strings:



Note that Android does not use zh-Hans and zh-Hant codes for Simplified and Traditional Chinese; instead, it only supports country-specific codes zh-CN and zh-TW.

To support different resolution images for high-density screens, create additional language folders with `-*dpi` suffixes, such as **drawables-es-mdpi**, **drawables-es-xdpi**, **drawables-es-xxdipi**, etc. See [Providing Alternative Android Resources](#) for more information.

### App Name

The contents of the **strings.xml** is just a single key-value to configure the app name:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ResxEspañol</string>
</resources>
```

Update the **MainActivity.cs** in the Android app project so that the `Label` references the strings XML.

```
[Activity (Label = "@string/app_name", MainLauncher = true,
    ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
```

The app now localizes the app name and image. Here's a screenshot of the result (in Spanish):

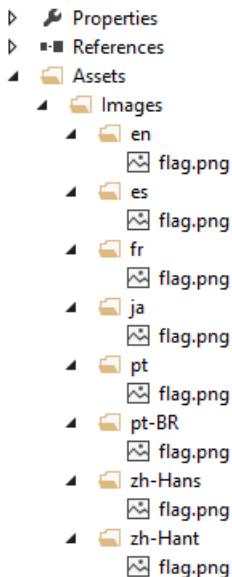


## Universal Windows Platform Application Projects

The Universal Windows Platform possesses a resource infrastructure that simplifies the localization of images and the app name. For more information about UWP localization, see [UWP Localization](#).

### Images

Images can be localized by placing them in a resource-specific folder, as demonstrated in the following screenshot:



At runtime the Windows resource infrastructure will select the appropriate image based on the user's locale.

## Summary

Xamarin.Forms applications can be localized using RESX files and .NET globalization classes. Apart from a small amount of platform-specific code to detect what language the user prefers, most of the localization effort is centralized in the common code.

Images are generally handled in a platform-specific way to take advantage of the multi-resolution support provided in both iOS and Android.

## Related Links

- [RESX Localization Sample](#)
- [TodoLocalized Sample App](#)
- [Cross-Platform Localization](#)
- [iOS Localization](#)
- [Android Localization](#)
- [UWP Localization](#)
- [Using the CultureInfo class \(MSDN\)](#)
- [Locating and Using Resources for a Specific Culture \(MSDN\)](#)

# Right-to-left localization

11/13/2018 • 4 minutes to read • [Edit Online](#)

*Right-to-left localization adds support for right-to-left flow direction to Xamarin.Forms applications.*

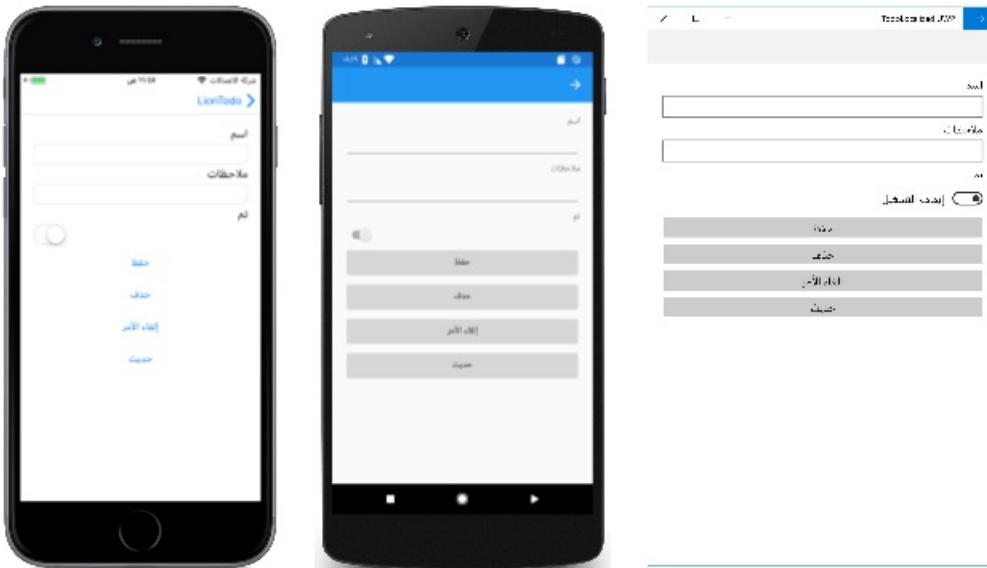
## NOTE

Right-to-left localization requires the use of iOS 9 or higher, and API 17 or higher on Android.

Flow direction is the direction in which the UI elements on the page are scanned by the eye. Some languages, such as Arabic and Hebrew, require that UI elements are laid out in a right-to-left flow direction. This can be achieved by setting the `VisualElement.FlowDirection` property. This property gets or sets the direction in which UI elements flow within any parent element that controls their layout, and should be set to one of the `FlowDirection` enumeration values:

- `LeftToRight`
- `RightToLeft`
- `MatchParent`

Setting the `FlowDirection` property to `RightToLeft` on an element generally sets the alignment to the right, the reading order to right-to-left, and the layout of the control to flow from right-to-left:



## TIP

You should only set the `FlowDirection` property on initial layout. Changing this value at runtime causes an expensive layout process that will affect performance.

The default `FlowDirection` property value for an element without a parent is `LeftToRight`, while the default `FlowDirection` for an element with a parent is `MatchParent`. Therefore, an element inherits the `FlowDirection` property value from its parent in the visual tree, and any element can override the value it gets from its parent.

## TIP

When localizing an app for right-to-left languages, set the `FlowDirection` property on a page or root layout. This causes all of the elements contained within the page, or root layout, to respond appropriately to the flow direction.

## Respecting device flow direction

Respecting the device's flow direction based on the selected language and region is an explicit developer choice and does not happen automatically. It can be achieved by setting the `FlowDirection` property on a page, or root layout, to the `static Device.FlowDirection` value:

```
<ContentPage ... FlowDirection="{x:Static Device.FlowDirection}"> />
```

```
this.FlowDirection = Device.FlowDirection;
```

All child elements of the page, or root layout, will by default then inherit the `Device.FlowDirection` value.

## Platform setup

Specific platform setup is required to enable right-to-left locales.

### iOS

The required right-to-left locale should be added as a supported language to the array items for the `CFBundleLocalizations` key in **Info.plist**. The following example shows Arabic having been added to the array for the `CFBundleLocalizations` key:

```
<key>CFBundleLocalizations</key>
<array>
    <string>en</string>
    <string>ar</string>
</array>
```

▼ Localizations	Array	(2 items)
	String	en
	String	ar

For more information, see [Localization Basics in iOS](#).

Right-to-left localization can then be tested by changing the language and region on the device/simulator to a right-to-left locale that was specified in **Info.plist**.

### WARNING

Please note that when changing the language and region to a right-to-left locale on iOS, any `DatePicker` views will throw an exception if you do not include the resources required for the locale. For example, when testing an app in Arabic that has a `DatePicker`, ensure that **mideast** is selected in the **Internationalization** section of the **iOS Build** pane.

### Android

The app's **AndroidManifest.xml** file should be updated so that the `<uses-sdk>` node sets the `android:minSdkVersion` attribute to 17, and the `<application>` node sets the `android:supportsRtl` attribute to `true`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <uses-sdk android:minSdkVersion="17" ... />
    <application ... android:supportsRtl="true">
        </application>
</manifest>
```

Right-to-left localization can then be tested by changing the device/emulator to use the right-to-left language, or by enabling **Force RTL layout direction** in **Settings > Developer Options**.

## Universal Windows Platform (UWP)

The required language resources should be specified in the `<Resources>` node of the **Package.appxmanifest** file.

The following example shows Arabic having been added to the `<Resources>` node:

```
<Resources>
    <Resource Language="x-generate"/>
    <Resource Language="en" />
    <Resource Language="ar" />
</Resources>
```

In addition, UWP requires that the app's default culture is explicitly defined in the .NET Standard library. This can be accomplished by setting the `NeutralResourcesLanguage` attribute in `AssemblyInfo.cs`, or in another class, to the default culture:

```
using System.Resources;

[assembly: NeutralResourcesLanguage("en")]
```

Right-to-left localization can then be tested by changing the language and region on the device to the appropriate right-to-left locale.

## Limitations

Xamarin.Forms right-to-left localization currently has a number of limitations:

- `NavigationPage` button location, toolbar item location, and transition animation is controlled by the device locale, rather than the `FlowDirection` property.
- `CarouselPage` swipe direction does not flip.
- `Image` visual content does not flip.
- `DisplayAlert` and `DisplayActionSheet` orientation is controlled by the device locale, rather than the `FlowDirection` property.
- `WebView` content does not respect the `FlowDirection` property.
- A `TextDirection` property needs to be added, to control text alignment.

## iOS

- `Stepper` orientation is controlled by the device locale, rather than the `FlowDirection` property.
- `EntryCell` text alignment is controlled by the device locale, rather than the `FlowDirection` property.
- `ContextActions` gestures and alignment are not reversed.

## Android

- `SearchBar` orientation is controlled by the device locale, rather than the `FlowDirection` property.
- `ContextActions` placement is controlled by the device locale, rather than the `FlowDirection` property.

## UWP

- `Editor` text alignment is controlled by the device locale, rather than the `FlowDirection` property.
- `FlowDirection` property is not inherited by `MasterDetailPage` children.
- `ContextActions` text alignment is controlled by the device locale, rather than the `FlowDirection` property.

## Right to left language support with Xamarin.University

[Xamarin.Forms 3.0 Right-to-Left Support, by Xamarin University](#)

## Related links

- [TodoLocalizedRTL Sample App](#)

# Xamarin.Forms Local Databases

10/10/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms supports database-driven applications using the SQLite database engine, which makes it possible to load and save objects in shared code. This article describes how Xamarin.Forms applications can read and write data to a local SQLite database using SQLite.Net.

## Overview

Xamarin.Forms applications can use the [SQLite.NET PCL NuGet](#) package to incorporate database operations into shared code by referencing the `SQLite` classes that ship in the NuGet. Database operations can be defined in the .NET Standard library project of the Xamarin.Forms solution.

The accompanying [sample application](#) is a simple Todo-list application. The following screenshots show how the sample appears on each platform:



## Using SQLite

To add SQLite support to a Xamarin.Forms .NET Standard library, use NuGet's search function to find **sqlite-net-pcl** and install the latest package:

The screenshot shows the NuGet Package Manager interface with the search bar containing "sqlite-net-pcl". On the left, there is a list of packages:

- sqlite-net-pcl** by Frank A. Krueger, v1.4.118  
SQLite-net Official Portable Library is the easy way to access sqlite from .NET...
- SQLite.Net-PCL** by Øystein Krog, Fr... v3.1.1  
A .NET client library to access SQLite embedded database files in a LINQ ma...
- sqlite-net-pcl-ecp** by Frank A. Krue... v1.3.3  
SQLite-net Portable Library is the easy way to access sqlite from .NET apps.
- SQLite-Net.Extensions**. v2017.530.810.15  
This package contains an extension for sqlite-net that...

On the right, the details for the **sqlite-net-pcl** package are shown:

**sqlite-net-pcl**  
Version: Latest stable 1.4.118   
  
**Description**  
SQLite-net is an open source and light weight library providing easy SQLite database storage for .NET, Mono, and Xamarin applications. This version uses SQLitePCLRaw to provide platform independent versions of SQLite.  
**Version:** 1.4.118  
**Author(s):** Frank A. Krueger

There are a number of NuGet packages with similar names, the correct package has these attributes:

- **Created by:** Frank A. Krueger

- **Id:** sqlite-net-pcl
- **NuGet link:** [sqlite-net-pcl](#)

#### NOTE

Despite the package name, use the **sqlite-net-pcl** NuGet package even in .NET Standard projects.

Once the reference has been added, add a property to the `App` class that returns a local file path for storing the database:

```
static TodoItemDatabase database;

public static TodoItemDatabase Database
{
    get
    {
        if (database == null)
        {
            database = new TodoItemDatabase(
                Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
                "TodoSQLite.db3"));
        }
        return database;
    }
}
```

The `TodoItemDatabase` constructor, which takes the path for the database file as an argument, is shown below:

```
public TodoItemDatabase(string dbPath)
{
    database = new SQLiteAsyncConnection(dbPath);
    database.CreateTableAsync<TodoItem>().Wait();
}
```

The advantage of exposing the database as a singleton is that a single database connection is created that's kept open while the application runs, therefore avoiding the expense of opening and closing the database file each time a database operation is performed.

The remainder of the `TodoItemDatabase` class contains SQLite queries that run cross-platform. Example query code is shown below (more details on the syntax can be found in [Using SQLite.NET with Xamarin.iOS](#).

```

public Task<List<TodoItem>> GetItemsAsync()
{
    return database.Table<TodoItem>().ToListAsync();
}

public Task<List<TodoItem>> GetItemsNotDoneAsync()
{
    return database.QueryAsync<TodoItem>("SELECT * FROM [TodoItem] WHERE [Done] = 0");
}

public Task<TodoItem> GetItemAsync(int id)
{
    return database.Table<TodoItem>().Where(i => i.ID == id).FirstOrDefaultAsync();
}

public Task<int> SaveItemAsync(TodoItem item)
{
    if (item.ID != 0)
    {
        return database.UpdateAsync(item);
    }
    else {
        return database.InsertAsync(item);
    }
}

public Task<int> DeleteItemAsync(TodoItem item)
{
    return database.DeleteAsync(item);
}

```

#### NOTE

The advantage of using the asynchronous SQLite.Net API is that database operations are moved to background threads. In addition, there's no need to write additional concurrency handling code because the API takes care of it.

## Summary

Xamarin.Forms supports database-driven applications using the SQLite database engine, which makes it possible to load and save objects in shared code.

This article focused on **accessing** a SQLite database using Xamarin.Forms. For more information on working with SQLite.Net itself, refer to the [SQLite.NET on Android](#) or [SQLite.NET on iOS](#) documentation.

## Related Links

- [Todo Sample](#)
- [Xamarin.Forms Samples](#)

# Xamarin.Forms MessagingCenter

10/31/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms includes a simple messaging service to send and receive messages.

## Overview

Xamarin.Forms `MessagingCenter` enables view models and other components to communicate with without having to know anything about each other besides a simple Message contract.

## How the MessagingCenter Works

There are two parts to `MessagingCenter`:

- **Subscribe** - Listen for messages with a certain signature and perform some action when they are received.  
Multiple subscribers can be listening for the same message.
- **Send** - Publish a message for listeners to act upon. If no listeners have subscribed then the message is ignored.

The `MessagingService` is a static class with `Subscribe` and `Send` methods that are used throughout the solution.

Messages have a string `message` parameter that is used as way to *address* messages. The `Subscribe` and `Send` methods use generic parameters to further control how messages are delivered - two messages with the same `message` text but different generic type arguments will not be delivered to the same subscriber.

The API for `MessagingCenter` is simple:

- `Subscribe<TSender> (object subscriber, string message, Action<TSender> callback, TSender source = null)`
- `Subscribe<TSender, TArgs> (object subscriber, string message, Action<TSender, TArgs> callback, TSender source = null)`
- `Send<TSender> (TSender sender, string message)`
- `Send<TSender, TArgs> (TSender sender, string message, TArgs args)`
- `Unsubscribe<TSender, TArgs> (object subscriber, string message)`
- `Unsubscribe<TSender> (object subscriber, string message)`

These methods are explained below.

## Using the MessagingCenter

Messages may be sent as a result of user-interaction (like a button click), a system event (like controls changing state) or some other incident (like an asynchronous download completing). Subscribers might be listening to change the appearance of the user interface, save data or trigger some other operation.

### Simple String Message

The simplest message contains just a string in the `message` parameter. A `Subscribe` method that *listens* for a simple string message is shown below - notice the generic type specifying the sender is expected to be of type `MainPage`. Any classes in the solution can subscribe to the message using this syntax:

```
MessagingCenter.Subscribe<MainPage> (this, "Hi", (sender) => {
    // do something whenever the "Hi" message is sent
});
```

In the `MainPage` class the following code *sends* the message. The `this` parameter is an instance of `MainPage`.

```
MessagingCenter.Send<MainPage> (this, "Hi");
```

The string doesn't change - it indicates the *message type* and is used for determining which subscribers to notify. This sort of message is used to indicate that some event occurred, such as "upload completed", where no further information is required.

### Passing an Argument

To pass an argument with the message, specify the argument Type in the `Subscribe` generic arguments and in the Action signature.

```
MessagingCenter.Subscribe<MainPage, string> (this, "Hi", (sender, arg) => {
    // do something whenever the "Hi" message is sent
    // using the 'arg' parameter which is a string
});
```

To send the message with argument, include the Type generic parameter and the value of the argument in the `Send` method call.

```
MessagingCenter.Send<MainPage, string> (this, "Hi", "John");
```

This simple example uses a `string` argument but any C# object can be passed.

### Unsubscribe

An object can unsubscribe from a message signature so that no future messages are delivered. The `Unsubscribe` method syntax should reflect the signature of the message (so may need to include the generic Type parameter for the message argument).

```
MessagingCenter.Unsubscribe<MainPage> (this, "Hi");
MessagingCenter.Unsubscribe<MainPage, string> (this, "Hi");
```

## Summary

The MessagingCenter is a simple way to reduce coupling, especially between view models. It can be used to send and receive simple messages or pass an argument between classes. Classes should unsubscribe from messages they no longer wish to receive.

## Related Links

- [MessagingCenterSample](#)
- [Xamarin.Forms Samples](#)

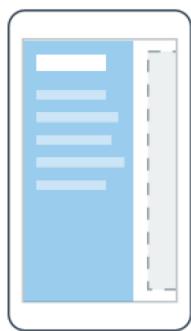
# Xamarin.Forms Navigation

7/12/2018 • 2 minutes to read • [Edit Online](#)

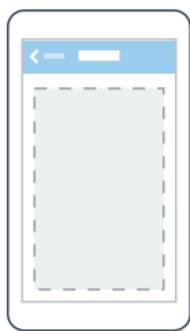
Xamarin.Forms provides a number of different page navigation experiences, depending upon the *Page* type being used.



ContentPage



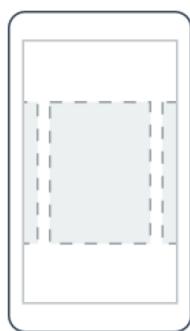
MasterDetailPage



NavigationPage



TabbedPage



CarouselPage

## Hierarchical Navigation

The `NavigationPage` class provides a hierarchical navigation experience where the user is able to navigate through pages, forwards and backwards, as desired. The class implements navigation as a last-in, first-out (LIFO) stack of `Page` objects.

## TabPage

The Xamarin.Forms `TabPage` consists of a list of tabs and a larger detail area, with each tab loading content into the detail area.

## CarouselPage

The Xamarin.Forms `CarouselPage` is a page that users can swipe from side to side to navigate through pages of content, like a gallery.

## MasterDetailPage

The Xamarin.Forms `MasterDetailPage` is a page that manages two pages of related information – a master page that presents items, and a detail page that presents details about items on the master page.

## Modal Pages

Xamarin.Forms also provides support for modal pages. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled.

## Displaying Pop-Ups

Xamarin.Forms provides two pop-up-like user interface elements: an alert and an action sheet. These interface elements can be used to ask users simple questions and to guide users through tasks.

# Hierarchical Navigation

9/20/2018 • 10 minutes to read • [Edit Online](#)

The `NavigationPage` class provides a hierarchical navigation experience where the user is able to navigate through pages, forwards and backwards, as desired. The class implements navigation as a last-in, first-out (LIFO) stack of `Page` objects. This article demonstrates how to use the `NavigationPage` class to perform navigation in a stack of pages.

To move from one page to another, an application will push a new page onto the navigation stack, where it will become the active page, as shown in the following diagram:



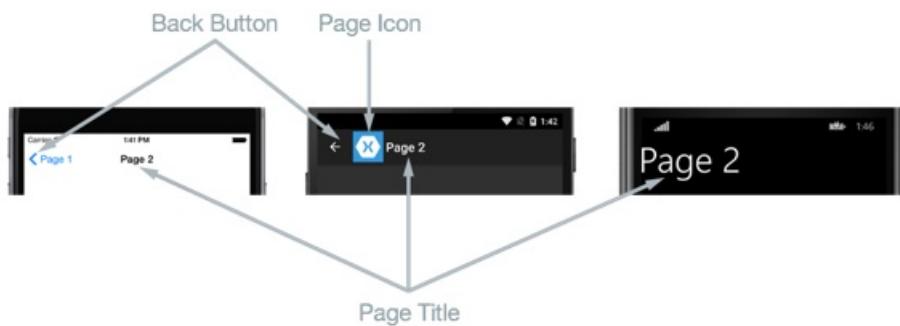
To return back to the previous page, the application will pop the current page from the navigation stack, and the new topmost page becomes the active page, as shown in the following diagram:



Navigation methods are exposed by the `Navigation` property on any `Page` derived types. These methods provide the ability to push pages onto the navigation stack, to pop pages from the navigation stack, and to perform stack manipulation.

## Performing Navigation

In hierarchical navigation, the `NavigationPage` class is used to navigate through a stack of `ContentPage` objects. The following screenshots show the main components of the `NavigationPage` on each platform:



The layout of a `NavigationPage` is dependent on the platform:

- On iOS, a navigation bar is present at the top of the page that displays a title, and that has a *Back* button that returns to the previous page.
- On Android, a navigation bar is present at the top of the page that displays a title, an icon, and a *Back* button that returns to the previous page. The icon is defined in the `[Activity]` attribute that decorates the `MainActivity` class in the Android platform-specific project.
- On the Universal Windows Platform, a navigation bar is present at the top of the page that displays a title.

On all the platforms, the value of the `Page.Title` property will be displayed as the page title.

**NOTE**

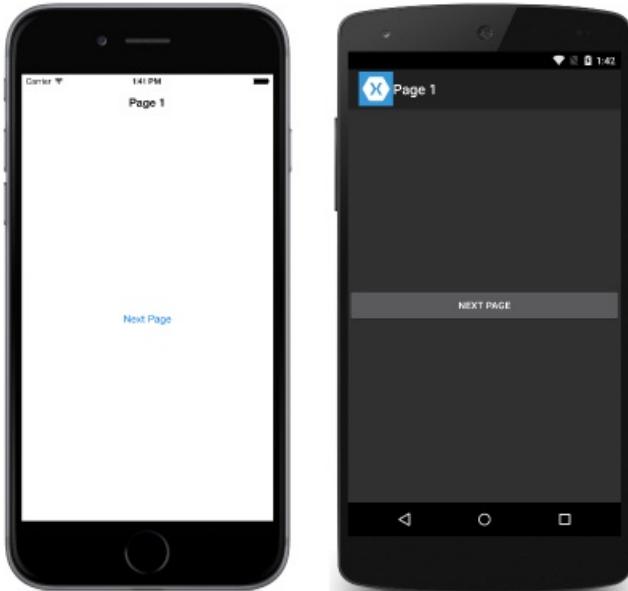
It's recommended that a `NavigationPage` should be populated with `ContentPage` instances only.

## Creating the Root Page

The first page added to a navigation stack is referred to as the *root* page of the application, and the following code example shows how this is accomplished:

```
public App ()  
{  
    MainPage = new NavigationPage (new Page1Xaml ());  
}
```

This causes the `Page1Xaml` `ContentPage` instance to be pushed onto the navigation stack, where it becomes the active page and the root page of the application. This is shown in the following screenshots:



**NOTE**

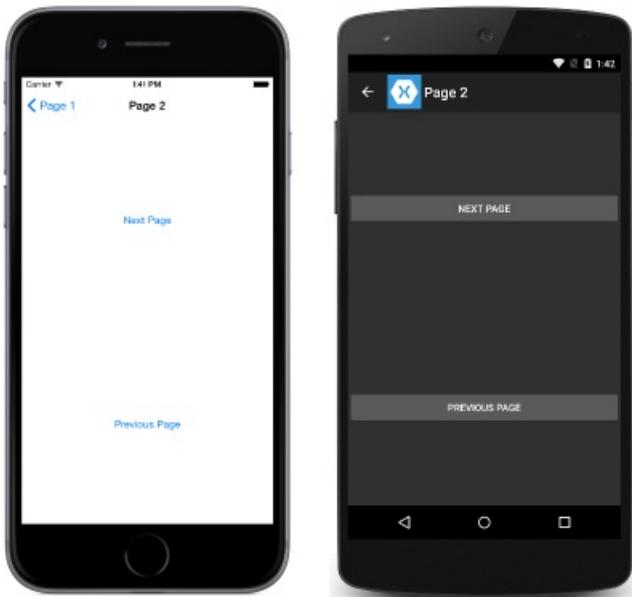
The `RootPage` property of a `NavigationPage` instance provides access to the first page in the navigation stack.

## Pushing Pages to the Navigation Stack

To navigate to `Page2Xaml`, it is necessary to invoke the `PushAsync` method on the `Navigation` property of the current page, as demonstrated in the following code example:

```
async void OnNextPageButtonClicked (object sender, EventArgs e)  
{  
    await Navigation.PushAsync (new Page2Xaml ());  
}
```

This causes the `Page2Xaml` instance to be pushed onto the navigation stack, where it becomes the active page. This is shown in the following screenshots:



When the `PushAsync` method is invoked, the following events occur:

- The page calling `PushAsync` has its `OnDisappearing` override invoked.
- The page being navigated to has its `OnAppearing` override invoked.
- The `PushAsync` task completes.

However, the precise order in which these events occur is platform dependent. For more information, see [Chapter 24](#) of Charles Petzold's *Xamarin.Forms* book.

#### NOTE

Calls to the `OnDisappearing` and `OnAppearing` overrides cannot be treated as guaranteed indications of page navigation. For example, on iOS, the `OnDisappearing` override is called on the active page when the application terminates.

## Popping Pages from the Navigation Stack

The active page can be popped from the navigation stack by pressing the *Back* button on the device, regardless of whether this is a physical button on the device or an on-screen button.

To programmatically return to the original page, the `Page2Xaml` instance must invoke the `PopAsync` method, as demonstrated in the following code example:

```
async void OnPreviousPageButtonClicked (object sender, EventArgs e)
{
    await Navigation.PopAsync ();
}
```

This causes the `Page2Xaml` instance to be removed from the navigation stack, with the new topmost page becoming the active page. When the `PopAsync` method is invoked, the following events occur:

- The page calling `PopAsync` has its `OnDisappearing` override invoked.
- The page being returned to has its `OnAppearing` override invoked.
- The `PopAsync` task returns.

However, the precise order in which these events occur is platform dependent. For more information see [Chapter 24](#) of Charles Petzold's *Xamarin.Forms* book.

As well as `PushAsync` and `PopAsync` methods, the `Navigation` property of each page also provides a

`PopToRootAsync` method, which is shown in the following code example:

```
async void OnRootPageButtonClicked (object sender, EventArgs e)
{
    await Navigation.PopToRootAsync ();
}
```

This method pops all but the root `Page` off the navigation stack, therefore making the root page of the application the active page.

### Animating Page Transitions

The `Navigation` property of each page also provides overridden push and pop methods that include a `boolean` parameter that controls whether to display a page animation during navigation, as shown in the following code example:

```
async void OnNextPageButtonClicked (object sender, EventArgs e)
{
    // Page appearance not animated
    await Navigation.PushAsync (new Page2Xaml (), false);
}

async void OnPreviousPageButtonClicked (object sender, EventArgs e)
{
    // Page appearance not animated
    await Navigation.PopAsync (false);
}

async void OnRootPageButtonClicked (object sender, EventArgs e)
{
    // Page appearance not animated
    await Navigation.PopToRootAsync (false);
}
```

Setting the `boolean` parameter to `false` disables the page-transition animation, while setting the parameter to `true` enables the page-transition animation, provided that it is supported by the underlying platform. However, the push and pop methods that lack this parameter enable the animation by default.

## Passing Data when Navigating

Sometimes it's necessary for a page to pass data to another page during navigation. Two techniques for accomplishing this are passing data through a page constructor, and by setting the new page's `BindingContext` to the data. Each will now be discussed in turn.

### Passing Data through a Page Constructor

The simplest technique for passing data to another page during navigation is through a page constructor parameter, which is shown in the following code example:

```
public App ()
{
    MainPage = new NavigationPage (new MainPage (DateTime.Now.ToString ("u")));
}
```

This code creates a `MainPage` instance, passing in the current date and time in ISO8601 format, which is wrapped in a `NavigationPage` instance.

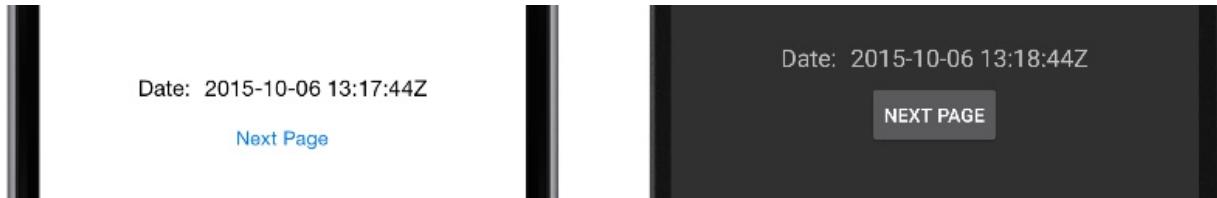
The `MainPage` instance receives the data through a constructor parameter, as shown in the following code example:

```

public MainPage (string date)
{
    InitializeComponent ();
    dateLabel.Text = date;
}

```

The data is then displayed on the page by setting the `Label.Text` property, as shown in the following screenshots:



## Passing Data through a BindingContext

An alternative approach for passing data to another page during navigation is by setting the new page's `BindingContext` to the data, as shown in the following code example:

```

async void OnNavigateButtonClicked (object sender, EventArgs e)
{
    var contact = new Contact {
        Name = "Jane Doe",
        Age = 30,
        Occupation = "Developer",
        Country = "USA"
    };

    var secondPage = new SecondPage ();
    secondPage.BindingContext = contact;
    await Navigation.PushAsync (secondPage);
}

```

This code sets the `BindingContext` of the `SecondPage` instance to the `Contact` instance, and then navigates to the `SecondPage`.

The `SecondPage` then uses data binding to display the `Contact` instance data, as shown in the following XAML code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="PassingData.SecondPage"
    Title="Second Page">
    <ContentPage.Content>
        <StackLayout HorizontalOptions="Center" VerticalOptions="Center">
            <StackLayout Orientation="Horizontal">
                <Label Text="Name:" HorizontalOptions="FillAndExpand" />
                <Label Text="{Binding Name}" FontSize="Medium" FontAttributes="Bold" />
            </StackLayout>
            ...
            <Button x:Name="navigateButton" Text="Previous Page" Clicked="OnNavigateButtonClicked" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

The following code example shows how the data binding can be accomplished in C#:

```

public class SecondPageCS : ContentPage
{
    public SecondPageCS ()
    {
        var nameLabel = new Label {
            FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
            FontAttributes = FontAttributes.Bold
        };
        nameLabel.SetBinding (Label.TextProperty, "Name");
        ...

        var navigateButton = new Button { Text = "Previous Page" };
        navigateButton.Clicked += OnNavigateButtonClicked;

        Content = new StackLayout {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Children = {
                new StackLayout {
                    Orientation = StackOrientation.Horizontal,
                    Children = {
                        new Label{ Text = "Name:", FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
HorizontalOptions = LayoutOptions.FillAndExpand },
                        nameLabel
                    }
                },
                ...
                navigateButton
            }
        };
    }

    async void OnNavigateButtonClicked (object sender, EventArgs e)
    {
        await Navigation.PopAsync ();
    }
}

```

The data is then displayed on the page by a series of `Label` controls, as shown in the following screenshots:



For more information about data binding, see [Data Binding Basics](#).

## Manipulating the Navigation Stack

The `Navigation` property exposes a `NavigationStack` property from which the pages in the navigation stack can be obtained. While Xamarin.Forms maintains access to the navigation stack, the `Navigation` property provides the `InsertPageBefore` and `RemovePage` methods for manipulating the stack by inserting pages or removing them.

The `InsertPageBefore` method inserts a specified page in the navigation stack before an existing specified page, as shown in the following diagram:



The `RemovePage` method removes the specified page from the navigation stack, as shown in the following diagram:



These methods enable a custom navigation experience, such as replacing a login page with a new page, following a successful login. The following code example demonstrates this scenario:

```
async void OnLoginButtonClicked (object sender, EventArgs e)
{
    ...
    var isValid = AreCredentialsCorrect (user);
    if (isValid) {
        App.IsUserLoggedIn = true;
        Navigation.InsertPageBefore (new MainPage (), this);
        await Navigation.PopAsync ();
    } else {
        // Login failed
    }
}
```

Provided that the user's credentials are correct, the `MainPage` instance is inserted into the navigation stack before the current page. The `PopAsync` method then removes the current page from the navigation stack, with the `MainPage` instance becoming the active page.

## Displaying Views in the Navigation Bar

Any Xamarin.Forms `View` can be displayed in the navigation bar of a `NavigationPage`. This is accomplished by setting the `NavigationPage.TitleView` attached property to a `View`. This attached property can be set on any `Page`, and when the `Page` is pushed onto a `NavigationPage`, the `NavigationPage` will respect the value of the property.

The following example, taken from the [Title View sample](#), shows how to set the `NavigationPage.TitleView` attached property from XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NavigationPageTitleView.TitleViewPager">
    <NavigationPage.TitleView>
        <Slider HeightRequest="44" WidthRequest="300" />
    </NavigationPage.TitleView>
    ...
</ContentPage>
```

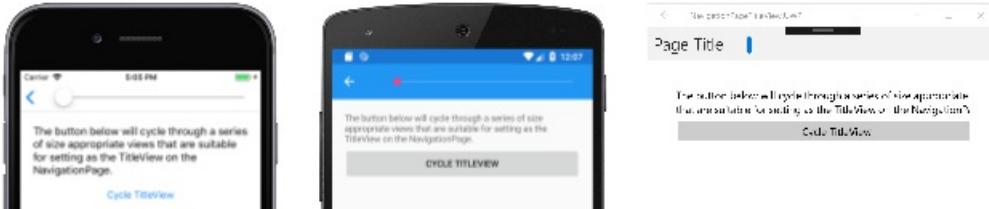
Here is the equivalent C# code:

```

public class TitleViewPage : ContentPage
{
    public TitleViewPage()
    {
        var titleView = new Slider { HeightRequest = 44, WidthRequest = 300 };
        NavigationPage.SetTitleView(this, titleView);
        ...
    }
}

```

This results in a `Slider` being displayed in the navigation bar on the `NavigationPage`:



### IMPORTANT

Many views won't appear in the navigation bar unless the size of the view is specified with the `WidthRequest` and `HeightRequest` properties. Alternatively, the view can be wrapped in a `StackLayout` with the `HorizontalOptions` and `VerticalOptions` properties set to appropriate values.

Note that, because the `Layout` class derives from the `View` class, the `TitleView` attached property can be set to display a layout class that contains multiple views. On iOS and the Universal Windows Platform (UWP), the height of the navigation bar can't be changed, and so clipping will occur if the view displayed in the navigation bar is larger than the default size of the navigation bar. However, on Android, the height of the navigation bar can be changed by setting the `NavigationPage.BarHeight` bindable property to a `double` representing the new height. For more information, see [Setting the Navigation Bar Height on a NavigationPage](#).

Alternatively, an extended navigation bar can be suggested by placing some of the content in the navigation bar, and some in a view at the top of the page content that you color match to the navigation bar. In addition, on iOS the separator line and shadow that's at the bottom of the navigation bar can be removed by setting the `NavigationPage.HideNavigationBarSeparator` bindable property to `true`. For more information, see [Hiding the Navigation Bar Separator on a NavigationPage](#).

### NOTE

The `BackButtonTitle`, `Title`, `TitleIcon`, and `TitleView` properties can all define values that occupy space on the navigation bar. While the navigation bar size varies by platform and screen size, setting all of these properties will result in conflicts due to the limited space available. Instead of attempting to use a combination of these properties, you may find that you can better achieve your desired navigation bar design by only setting the `TitleView` property.

### Limitations

There are a number of limitations to be aware of when displaying a `View` in the navigation bar of a `NavigationPage`:

- On iOS, views placed in the navigation bar of a `NavigationPage` appear in a different position depending on whether large titles are enabled. For more information about enabling large titles, see [Displaying Large Titles](#).
- On Android, placing views in the navigation bar of a `NavigationPage` can only be accomplished in apps that use app-compat.
- It's not recommended to place large and complex views, such as `ListView` and `TableView`, in the navigation

bar of a [NavigationPage](#).

## Related Links

- [Page Navigation](#)
- [Hierarchical \(sample\)](#)
- [PassingData \(sample\)](#)
- [LoginFlow \(sample\)](#)
- [TitleView \(sample\)](#)
- [How to Create a Sign In Screen Flow in Xamarin.Forms \(Xamarin University Video\) Sample](#)
- [How to Create a Sign In Screen Flow in Xamarin.Forms \(Xamarin University Video\)](#)
- [NavigationPage](#)

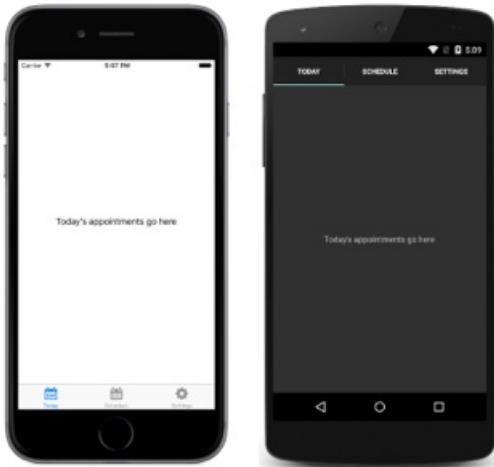
# Xamarin.Forms Tabbed Page

11/20/2018 • 5 minutes to read • [Edit Online](#)

The `Xamarin.Forms TabbedPage` consists of a list of tabs and a larger detail area, with each tab loading content into the detail area. This article demonstrates how to use a `TabbedPage` to navigate through a collection of pages.

## Overview

The following screenshots show a `TabbedPage` on each platform:



The following screenshots focus on the tab format on each platform:



The layout of a `TabbedPage`, and its tabs, is dependent on the platform:

- On iOS, the list of tabs appears at the bottom of the screen, and the detail area is above. Each tab also has an icon image which should be a 30x30 PNG with transparency for normal resolution, 60x60 for high resolution, and 90x90 for iPhone 6 Plus resolution. If there are more than five tabs, a `More` tab will appear, which can be used to access the additional tabs. For more information about loading images in a `Xamarin.Forms` application, see [Working with Images](#). For more information about icon requirements, see [Creating Tabbed Applications](#).

### NOTE

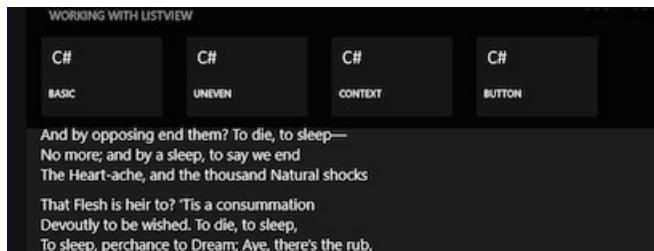
Note that the `TabbedRenderer` for iOS has an overridable `GetIcon` method that can be used to load tab icons from a specified source. This override makes it possible to use SVG images as icons on a `TabbedPage`. In addition, selected and unselected versions of an icon can be provided.

- On Android, the list of tabs appears at the top of the screen by default, and the detail area is below. However, the tab list can be moved to the bottom of the screen with a platform-specific. For more information, see [Setting TabbedPage Toolbar Placement and Color](#).

#### NOTE

Note that when using AppCompat on Android, each tab will also display an icon. In addition, the `TabPageRenderer` for Android AppCompat has an overridable `GetIconDrawable` method that can be used to load tab icons from a custom `Drawable`. This override makes it possible to use SVG images as icons on a `TabPage`, and works with both top and bottom tab bars. Alternatively, the overridable `SetTabIcon` method can be used to load tab icons from a custom `Drawable` for top tab bars.

- On Windows tablet form-factors, the tabs aren't always visible and users need to swipe-down (or right-click, if they have a mouse attached) to view the tabs in a `TabPage` (as shown below).



## Creating a TabbedPage

Two approaches can be used to create a `TabPage`:

- Populate the `TabPage` with a collection of child `Page` objects, such as a collection of `ContentPage` instances.
- Assign a collection to the `ItemsSource` property and assign a `DataTemplate` to the `ItemTemplate` property to return pages for objects in the collection.

With both approaches, the `TabPage` will display each page as the user selects each tab.

#### NOTE

It's recommended that a `TabPage` should be populated with `NavigationPage` and `ContentPage` instances only. This will help to ensure a consistent user experience across all platforms.

### Populating a TabbedPage with a Page Collection

The following XAML code example shows a `TabPage` constructed by populating it with a collection of child `Page` objects:

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:TabPageWithNavigationPage;assembly=TabPageWithNavigationPage"
    x:Class="TabPageWithNavigationPage.MainPage">
    <local:TodayPage />
    <NavigationView Title="Schedule" Icon="schedule.png">
        <x:Arguments>
            <local:SchedulePage />
        </x:Arguments>
    </NavigationView>
</TabbedPage>
```

The following code example shows the equivalent `TabPage` created in C#:

```
public class MainPageCS : TabbedPage
{
    public MainPageCS ()
    {
        var navigationPage = new NavigationPage (new SchedulePageCS ());
        navigationPage.Icon = "schedule.png";
        navigationPage.Title = "Schedule";

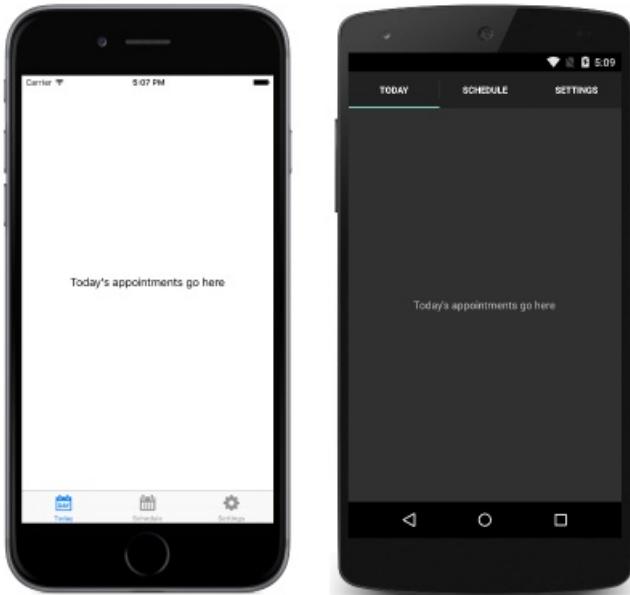
        Children.Add (new TodayPageCS ());
        Children.Add (navigationPage);
    }
}
```

The `TabbedPage` is populated with two child `Page` objects. The first child is a `ContentPage` instance, and the second tab is a `NavigationPage` containing a `ContentPage` instance.

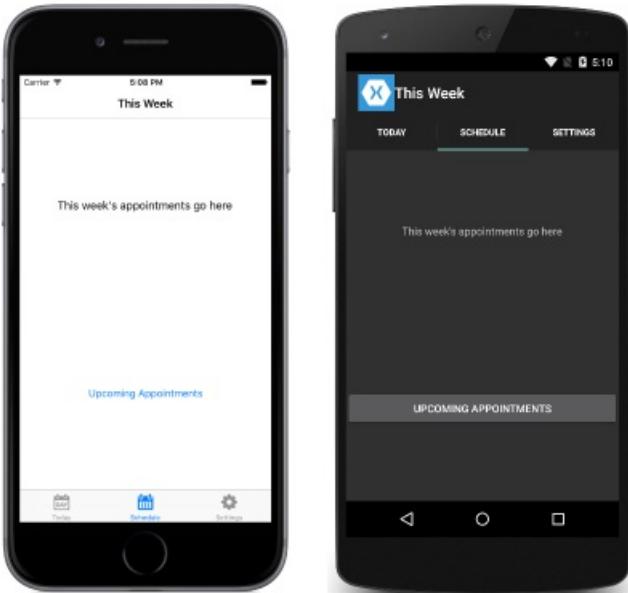
**NOTE**

The `TabbedPage` does not support UI virtualization. Therefore, performance may be affected if the `TabbedPage` contains too many child elements.

The following screenshots show the `TodayPage` `ContentPage` instance, which is shown on the *Today* tab:



Selecting the *Schedule* tab displays the `SchedulePage` `ContentPage` instance, which is wrapped in a `NavigationPage` instance, and is shown in the following screenshot:



For information about the layout of a [NavigationPage](#), see [Performing Navigation](#).

#### NOTE

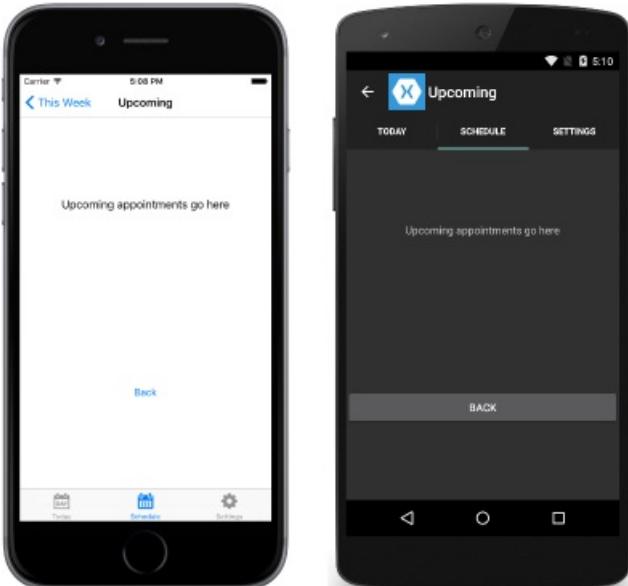
While it's acceptable to place a [NavigationPage](#) into a [TabbedPage](#), it's not recommended to place a [TabbedPage](#) into a [NavigationPage](#). This is because, on iOS, a [UITabBarController](#) always acts as a wrapper for the [UINavigationController](#). For more information, see [Combined View Controller Interfaces](#) in the iOS Developer Library.

#### Navigation Inside a Tab

Navigation can be performed from the second tab by invoking the [PushAsync](#) method on the [Navigation](#) property of the [ContentPage](#) instance, as demonstrated in the following code example:

```
async void OnUpcomingAppointmentsButtonClicked (object sender, EventArgs e)
{
    await Navigation.PushAsync (new UpcomingAppointmentsPage ());
}
```

This causes the [UpcomingAppointmentsPage](#) instance to be pushed onto the navigation stack, where it becomes the active page. This is shown in the following screenshots:



For more information about performing navigation using the [NavigationPage](#) class, see [Hierarchical Navigation](#).

## Populating a TabbedPage with a Template

The following XAML code example shows a [TabbedPage](#) constructed by assigning a [DataTemplate](#) to the [ItemTemplate](#) property to return pages for objects in the collection:

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:TabPageDemo;assembly=TabPageDemo"
    x:Class="TabPageDemo.TabPageDemoPage">
    <TabbedPage.Resources>
        <ResourceDictionary>
            <local:NonNullToBooleanConverter x:Key="booleanConverter" />
        </ResourceDictionary>
    </TabbedPage.Resources>
    <TabbedPage.ItemTemplate>
        <DataTemplate>
            <ContentPage Title="{Binding Name}" Icon="monkeyicon.png">
                <StackLayout Padding="5, 25">
                    <Label Text="{Binding Name}" Font="Bold, Large" HorizontalOptions="Center" />
                    <Image Source="{Binding PhotoUrl}" WidthRequest="200" HeightRequest="200" />
                    <StackLayout Padding="50, 10">
                        <StackLayout Orientation="Horizontal">
                            <Label Text="Family:" HorizontalOptions="FillAndExpand" />
                            <Label Text="{Binding Family}" Font="Bold, Medium" />
                        </StackLayout>
                        ...
                    </StackLayout>
                </StackLayout>
            </ContentPage>
        </DataTemplate>
    </TabbedPage.ItemTemplate>
</TabbedPage>
```

The [TabbedPage](#) is populated with data by setting the [ItemsSource](#) property in the constructor for the code-behind file:

```
public TabbedPageDemoPage ()
{
    ...
    ItemsSource = MonkeyDataModel.All;
}
```

The following code example shows the equivalent [TabbedPage](#) created in C#:

```

public class TabbedPageDemoPageCS : TabbedPage
{
    public TabbedPageDemoPageCS ()
    {
        var booleanConverter = new NonNullToBooleanConverter ();

        ItemTemplate = new DataTemplate (() => {
            var nameLabel = new Label {
                FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label)),
                FontAttributes = FontAttributes.Bold,
                HorizontalOptions = LayoutOptions.Center
            };
            nameLabel.SetBinding (Label.TextProperty, "Name");

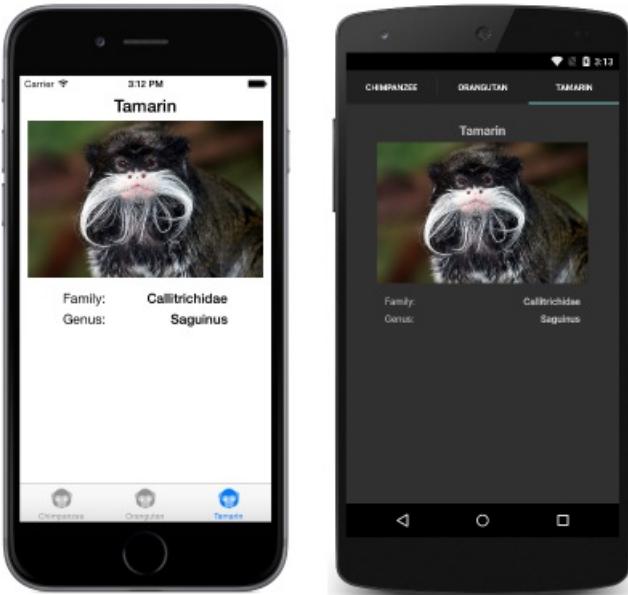
            var image = new Image { WidthRequest = 200, HeightRequest = 200 };
            image.SetBinding (Image.SourceProperty, "PhotoUrl");

            var familyLabel = new Label {
                FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                FontAttributes = FontAttributes.Bold
            };
            familyLabel.SetBinding (Label.TextProperty, "Family");
            ...

            var contentPage = new ContentPage {
                Icon = "monkeyicon.png",
                Content = new StackLayout {
                    Padding = new Thickness (5, 25),
                    Children = {
                        nameLabel,
                        image,
                        new StackLayout {
                            Padding = new Thickness (50, 10),
                            Children = {
                                new StackLayout {
                                    Orientation = StackOrientation.Horizontal,
                                    Children = {
                                        new Label { Text = "Family:", HorizontalOptions = LayoutOptions.FillAndExpand },
                                        familyLabel
                                    }
                                },
                                ...
                            }
                        }
                    }
                };
                contentPage.SetBinding (TitleProperty, "Name");
                return contentPage;
            });
            ItemsSource = MonkeyDataModel.All;
        }
    }
}

```

Each tab displays a `ContentPage` that uses a series of `StackLayout` and `Label` instances to display data for the tab. The following screenshots show the content for the *Tamarin* tab:



Selecting another tab then displays content for that tab.

#### NOTE

The `TabPage` does not support UI virtualization. Therefore, performance may be affected if the `TabPage` contains too many child elements.

For more information about the `TabPage`, see [Chapter 25](#) of Charles Petzold's Xamarin.Forms book.

## Summary

This article demonstrated how to use a `TabPage` to navigate through a collection of pages. The `Xamarin.Forms TabbedPage` consists of a list of tabs and a larger detail area, with each tab loading content into the detail area.

## Related Links

- [Page Varieties](#)
- [TabPageWithNavigationPage \(sample\)](#)
- [TabPage \(sample\)](#)
- [TabPage](#)

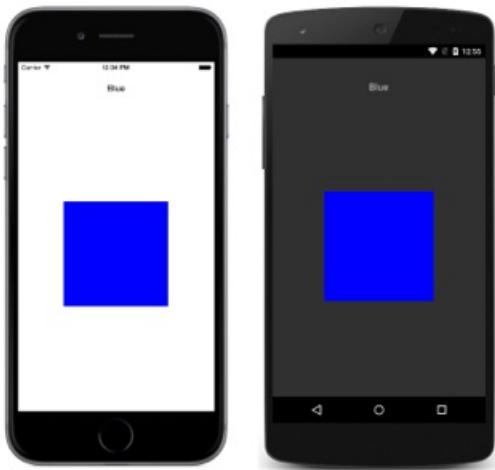
# Xamarin.Forms Carousel Page

10/12/2018 • 4 minutes to read • [Edit Online](#)

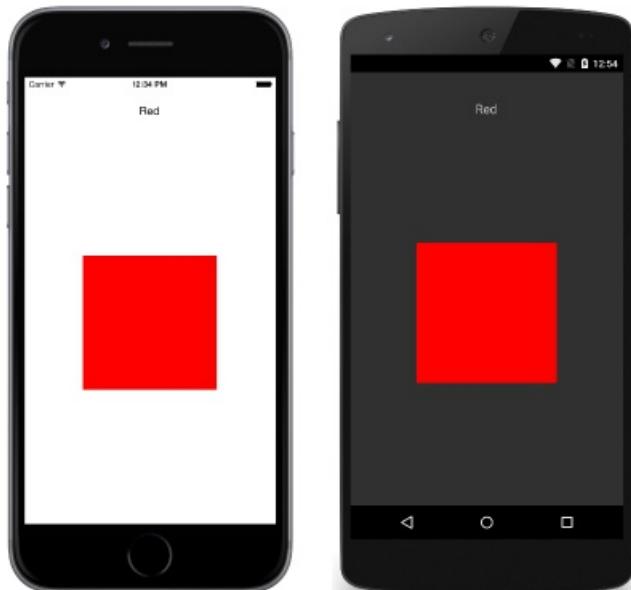
The `Xamarin.Forms CarouselPage` is a page that users can swipe from side to side to navigate through pages of content, like a gallery. This article demonstrates how to use a `CarouselPage` to navigate through a collection of pages.

## Overview

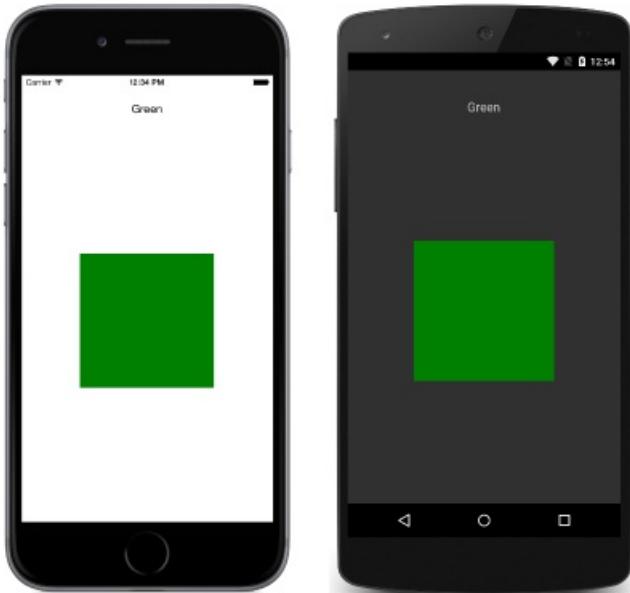
The following screenshots show a `CarouselPage` on each platform:



The layout of a `CarouselPage` is identical on each platform. Pages can be navigated through by swiping right to left to navigate forwards through the collection, and by swiping left to right to navigate backwards through the collection. The following screenshots show the first page in a `CarouselPage` instance:



Swiping from right to left moves to the second page, as shown in the following screenshots:



Swiping from right to left again moves to the third page, while swiping from left to right returns to the previous page.

## Creating a CarouselPage

Two approaches can be used to create a `CarouselPage`:

- **Populate** the `CarouselPage` with a collection of child `ContentPage` instances.
- **Assign** a collection to the `ItemsSource` property and assign a `DataTemplate` to the `ItemTemplate` property to return `ContentPage` instances for objects in the collection.

With both approaches, the `CarouselPage` will then display each page in turn, with a swipe interaction moving to the next page to be displayed.

### NOTE

A `CarouselPage` can only be populated with `ContentPage` instances, or `ContentPage` derivatives.

### Populating a CarouselPage with a Page Collection

The following XAML code example shows a `CarouselPage` that displays three `ContentPage` instances:

```
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="CarouselPageNavigation.MainPage">
    <ContentPage>
        <ContentPage.Padding>
            <OnPlatform x:TypeArguments="Thickness">
                <On Platform="iOS, Android" Value="0,40,0,0" />
            </OnPlatform>
        </ContentPage.Padding>
        <StackLayout>
            <Label Text="Red" FontSize="Medium" HorizontalOptions="Center" />
            <BoxView Color="Red" WidthRequest="200" HeightRequest="200" HorizontalOptions="Center"
VerticalOptions="CenterAndExpand" />
        </StackLayout>
    </ContentPage>
    <ContentPage>
        ...
    </ContentPage>
    <ContentPage>
        ...
    </ContentPage>
</CarouselPage>
```

The following code example shows the equivalent UI in C#:

```

public class MainPageCS : CarouselPage
{
    public MainPageCS ()
    {
        Thickness padding;
        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
            case Device.Android:
                padding = new Thickness(0, 40, 0, 0);
                break;
            default:
                padding = new Thickness();
                break;
        }

        var redContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                Children = {
                    new Label {
                        Text = "Red",
                        FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                        HorizontalOptions = LayoutOptions.Center
                    },
                    new BoxView {
                        Color = Color.Red,
                        WidthRequest = 200,
                        HeightRequest = 200,
                        HorizontalOptions = LayoutOptions.Center,
                        VerticalOptions = LayoutOptions.CenterAndExpand
                    }
                }
            }
        };
        var greenContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                ...
            }
        };
        var blueContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                ...
            }
        };

        Children.Add (redContentPage);
        Children.Add (greenContentPage);
        Children.Add (blueContentPage);
    }
}

```

Each `ContentPage` simply displays a `Label` for a particular color and a `BoxView` of that color.

#### NOTE

The `CarouselPage` does not support UI virtualization. Therefore, performance may be affected if the `CarouselPage` contains too many child elements.

If a `CarouselPage` is embedded into the `Detail` page of a `MasterDetailPage`, the `MasterDetailPage.IsGestureEnabled` property should be set to `false` to prevent gesture conflicts between the

`CarouselPage` and the `MasterDetailPage`.

For more information about the `CarouselPage`, see [Chapter 25](#) of Charles Petzold's Xamarin.Forms book.

### Populating a `CarouselPage` with a Template

The following XAML code example shows a `CarouselPage` constructed by assigning a `DataTemplate` to the `ItemTemplate` property to return pages for objects in the collection:

```
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="CarouselPageNavigation.MainPage">
    <CarouselPage.ItemTemplate>
        <DataTemplate>
            <ContentPage>
                <ContentPage.Padding>
                    <OnPlatform x:TypeArguments="Thickness">
                        <On Platform="iOS, Android" Value="0,40,0,0" />
                    </OnPlatform>
                </ContentPage.Padding>
                <StackLayout>
                    <Label Text="{Binding Name}" FontSize="Medium" HorizontalOptions="Center" />
                    <BoxView Color="{Binding Color}" WidthRequest="200" HeightRequest="200"
HorizontalOptions="Center" VerticalOptions="CenterAndExpand" />
                </StackLayout>
            </ContentPage>
        </DataTemplate>
    </CarouselPage.ItemTemplate>
</CarouselPage>
```

The `CarouselPage` is populated with data by setting the `ItemsSource` property in the constructor for the code-behind file:

```
public MainPage ()
{
    ...
    ItemsSource = ColorsDataModel.All;
}
```

The following code example shows the equivalent `CarouselPage` created in C#:

```

public class MainPageCS : CarouselPage
{
    public MainPageCS ()
    {
        Thickness padding;
        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
            case Device.Android:
                padding = new Thickness(0, 40, 0, 0);
                break;
            default:
                padding = new Thickness();
                break;
        }

        ItemTemplate = new DataTemplate (() => {
            var nameLabel = new Label {
                FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                HorizontalOptions = LayoutOptions.Center
            };
            nameLabel.SetBinding (Label.TextProperty, "Name");

            var colorBoxView = new BoxView {
                WidthRequest = 200,
                HeightRequest = 200,
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.CenterAndExpand
            };
            colorBoxView.SetBinding (BoxView.ColorProperty, "Color");

            return new ContentPage {
                Padding = padding,
                Content = new StackLayout {
                    Children = {
                        nameLabel,
                        colorBoxView
                    }
                }
            };
        });

        ItemsSource = ColorsDataModel.All;
    }
}

```

Each `ContentPage` simply displays a `Label` for a particular color and a `BoxView` of that color.

#### NOTE

The `CarouselPage` does not support UI virtualization. Therefore, performance may be affected if the `CarouselPage` contains too many child elements.

If a `CarouselPage` is embedded into the `Detail` page of a `MasterDetailPage`, the `MasterDetailPage.IsGestureEnabled` property should be set to `false` to prevent gesture conflicts between the `CarouselPage` and the `MasterDetailPage`.

For more information about the `CarouselPage`, see [Chapter 25](#) of Charles Petzold's Xamarin.Forms book.

## Summary

This article demonstrated how to use a `CarouselPage` to navigate through a collection of pages. The `CarouselPage`

is a page that users can swipe from side to side to navigate through pages of content,much like a gallery.

## Related Links

- [Page Varieties](#)
- [CarouselPage \(sample\)](#)
- [CarouselPageTemplate \(sample\)](#)
- [CarouselPage](#)

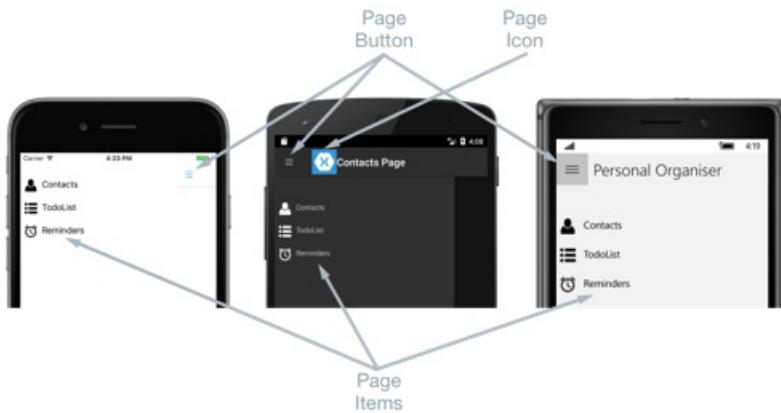
# Xamarin.Forms Master-Detail Page

10/31/2018 • 8 minutes to read • [Edit Online](#)

The `Xamarin.Forms MasterDetailPage` is a page that manages two related pages of information – a master page that presents items, and a detail page that presents details about items on the master page. This article explains how to use a `MasterDetailPage` and navigate between its pages of information.

## Overview

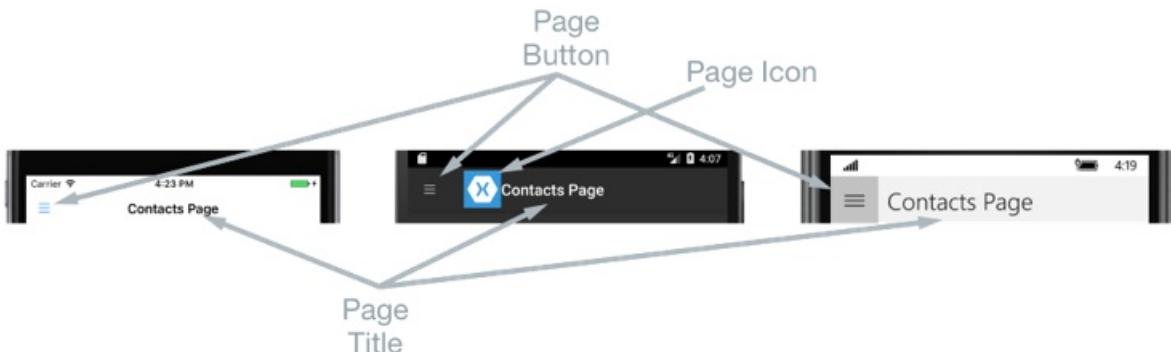
A master page typically displays a list of items, as shown in the following screenshots:



The location of the list of items is identical on each platform, and selecting one of the items will navigate to the corresponding detail page. In addition, the master page also features a navigation bar that contains a button that can be used to navigate to the active detail page:

- On iOS, the navigation bar is present at the top of the page and has a button that navigates to the detail page. In addition, the active detail page can be navigated to by swiping the master page to the left.
- On Android, the navigation bar is present at the top of the page and displays a title, an icon, and a button that navigates to the detail page. The icon is defined in the `[Activity]` attribute that decorates the `MainActivity` class in the Android platform-specific project. In addition, the active detail page can be navigated to by swiping the master page to the left, by tapping the detail page at the far right of the screen, and by tapping the `Back` button at the bottom of the screen.
- On the Universal Windows Platform (UWP), the navigation bar is present at the top of the page and has a button that navigates to the detail page.

A detail page displays data that corresponds to the item selected on the master page, and the main components of the detail page are shown in the following screenshots:



The detail page contains a navigation bar, whose contents are platform-dependent:

- On iOS, the navigation bar is present at the top of the page and displays a title, and has a button that returns to the master page, provided that the detail page instance is wrapped in the `NavigationPage` instance. In addition, the master page can be returned to by swiping the detail page to the right.
- On Android, a navigation bar is present at the top of the page and displays a title, an icon, and a button that returns to the master page. The icon is defined in the `[Activity]` attribute that decorates the `MainActivity` class in the Android platform-specific project.
- On UWP, the navigation bar is present at the top of the page and displays a title, and has a button that returns to the master page.

## Navigation Behavior

The behavior of the navigation experience between master and detail pages is platform dependent:

- On iOS, the detail page *slides* to the right as the master page slides from the left, and the left part of the detail page is still visible.
- On Android, the detail and master pages are *overlaid* on each other.
- On UWP, the detail and master pages are *swapped*.

Similar behavior will be observed in landscape mode, except that the master page on iOS and Android has a similar width as the master page in portrait mode, so more of the detail page will be visible.

For information about controlling the navigation behavior, see [Controlling the Detail Page Display Behavior](#).

## Creating a MasterDetailPage

A `MasterDetailPage` contains `Master` and `Detail` properties that are both of type `Page`, which are used to get and set the master and detail pages respectively.

### IMPORTANT

A `MasterDetailPage` is designed to be a root page, and using it as a child page in other page types could result in unexpected and inconsistent behavior. In addition, it's recommended that the master page of a `MasterDetailPage` should always be a `ContentPage` instance, and that the detail page should only be populated with `TabbedPage`, `NavigationPage`, and `ContentPage` instances. This will help to ensure a consistent user experience across all platforms.

The following XAML code example shows a `MasterDetailPage` that sets the `Master` and `Detail` properties:

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MasterDetailPageNavigation;assembly=MasterDetailPageNavigation"
    x:Class="MasterDetailPageNavigation.MainPage">
    <MasterDetailPage.Master>
        <local:MasterPage x:Name="masterPage" />
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <NavigationPage>
            <x:Arguments>
                <local:ContactsPage />
            </x:Arguments>
        </NavigationPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

The following code example shows the equivalent `MasterDetailPage` created in C#:

```

public class MainPageCS : MasterDetailPage
{
    MasterPageCS masterPage;

    public MainPageCS ()
    {
        masterPage = new MasterPageCS ();
        Master = masterPage;
        Detail = new NavigationPage (new ContactsPageCS ());
        ...
    }
    ...
}

```

The `MasterDetailPage.Master` property is set to a `ContentPage` instance. The `MasterDetailPage.Detail` property is set to a `NavigationPage` containing a `ContentPage` instance.

## Creating the Master Page

The following XAML code example shows the declaration of the `MasterPage` object, which is referenced through the `MasterDetailPage.Master` property:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="using:MasterDetailPageNavigation"
    x:Class="MasterDetailPageNavigation.MasterPage"
    Padding="0,40,0,0"
    Icon="hamburger.png"
    Title="Personal Organiser">
    <StackLayout>
        <ListView x:Name="listView" x:FieldModifier="public">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:MasterPageItem}">
                    <local:MasterPageItem Title="Contacts" IconSource="contacts.png" TargetType="{x:Type
local:ContactsPage}" />
                    <local:MasterPageItem Title="TodoList" IconSource="todo.png" TargetType="{x:Type
local:TodoListPage}" />
                    <local:MasterPageItem Title="Reminders" IconSource="reminders.png" TargetType="{x:Type
local:ReminderPage}" />
                </x:Array>
            </ListView.ItemsSource>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <Grid Padding="5,10">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="30"/>
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>
                            <Image Source="{Binding IconSource}" />
                            <Label Grid.Column="1" Text="{Binding Title}" />
                        </Grid>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>

```

The page consists of a `ListView` that's populated with data in XAML by setting its `ItemsSource` property to an array of `MasterPageItem` instances. Each `MasterPageItem` defines `Title`, `IconSource`, and `TargetType` properties.

A `DataTemplate` is assigned to the `ListView.ItemTemplate` property, to display each `MasterPageItem`. The

`DataTemplate` contains a `ViewCell` that consists of an `Image` and a `Label`. The `Image` displays the `IconSource` property value, and the `Label` displays the `Title` property value, for each `MasterPageItem`.

The page has its `Title` and `Icon` properties set. The icon will appear on the detail page, provided that the detail page has a title bar. This must be enabled on iOS by wrapping the detail page instance in a `NavigationPage` instance.

**NOTE**

The `MasterDetailPage.Master` page must have its `Title` property set, or an exception will occur.

The following code example shows the equivalent page created in C#:

```

public class MasterPageCS : ContentPage
{
    public ListView ListView { get { return listView; } }

    ListView listView;

    public MasterPageCS ()
    {
        var masterPageItems = new List<MasterPageItem> ();
        masterPageItems.Add (new MasterPageItem {
            Title = "Contacts",
            IconSource = "contacts.png",
            TargetType = typeof(ContactsPageCS)
        });
        masterPageItems.Add (new MasterPageItem {
            Title = "TodoList",
            IconSource = "todo.png",
            TargetType = typeof(TodoListPageCS)
        });
        masterPageItems.Add (new MasterPageItem {
            Title = "Reminders",
            IconSource = "reminders.png",
            TargetType = typeof(ReminderPageCS)
        });

        listView = new ListView {
            ItemsSource = masterPageItems,
            ItemTemplate = new DataTemplate (() => {
                var grid = new Grid { Padding = new Thickness(5, 10) };
                grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(30) });
                grid.ColumnDefinitions.Add(new ColumnDefinition { Width = GridLength.Star });

                var image = new Image();
                image.SetBinding(Image.SourceProperty, "IconSource");
                var label = new Label { VerticalOptions = LayoutOptions.FillAndExpand };
                label.SetBinding(Label.TextProperty, "Title");

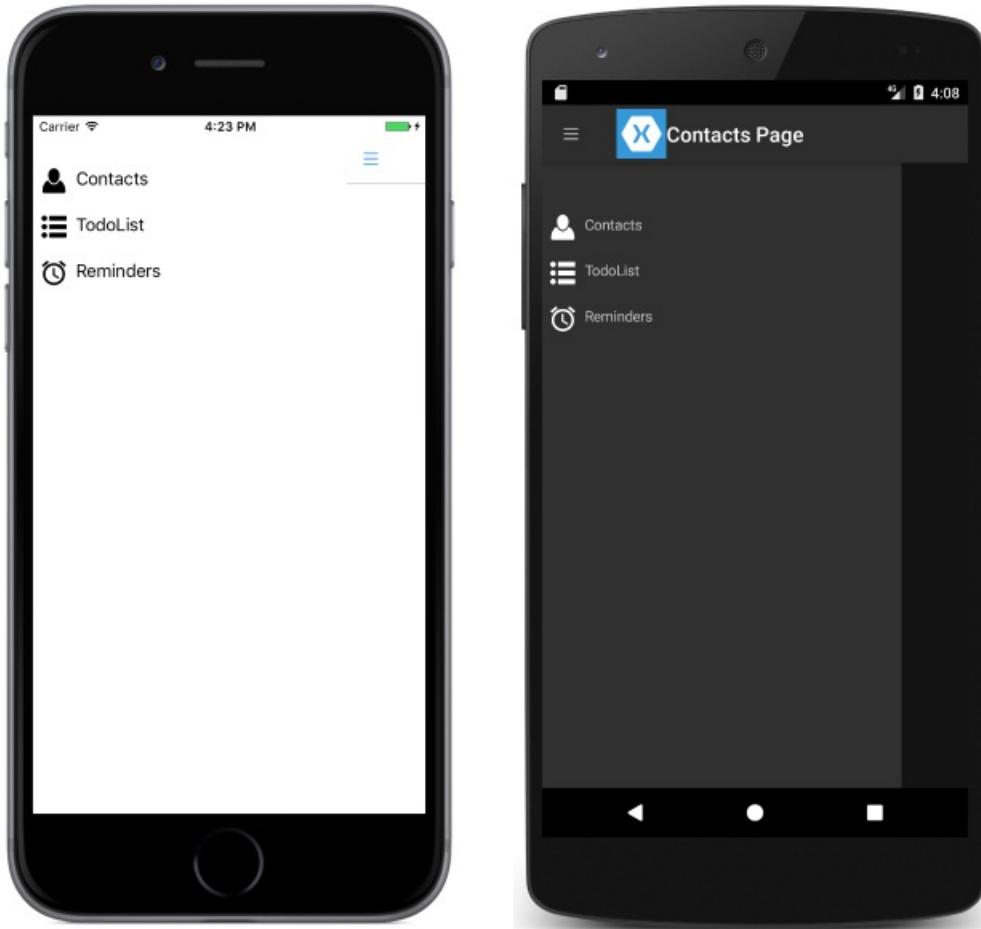
                grid.Children.Add(image);
                grid.Children.Add(label, 1, 0);

                return new ViewCell { View = grid };
            }),
            SeparatorVisibility = SeparatorVisibility.None
        };

        Icon = "hamburger.png";
        Title = "Personal Organiser";
        Content = new StackLayout
        {
            Children = { listView }
        };
    }
}

```

The following screenshots show the master page on each platform:



### Creating and Displaying the Detail Page

The `MasterPage` instance contains a `ListView` property that exposes its `ListView` instance so that the `MasterDetailPage` instance can register an event-handler to handle the `ItemSelected` event. This enables the `MainPage` instance to set the `Detail` property to the page that represents the selected `ListView` item. The following code example shows the event-handler:

```
public partial class MainPage : MasterDetailPage
{
    public MainPage ()
    {
        ...
        masterPage.listView.ItemSelected += OnItemSelected;
    }

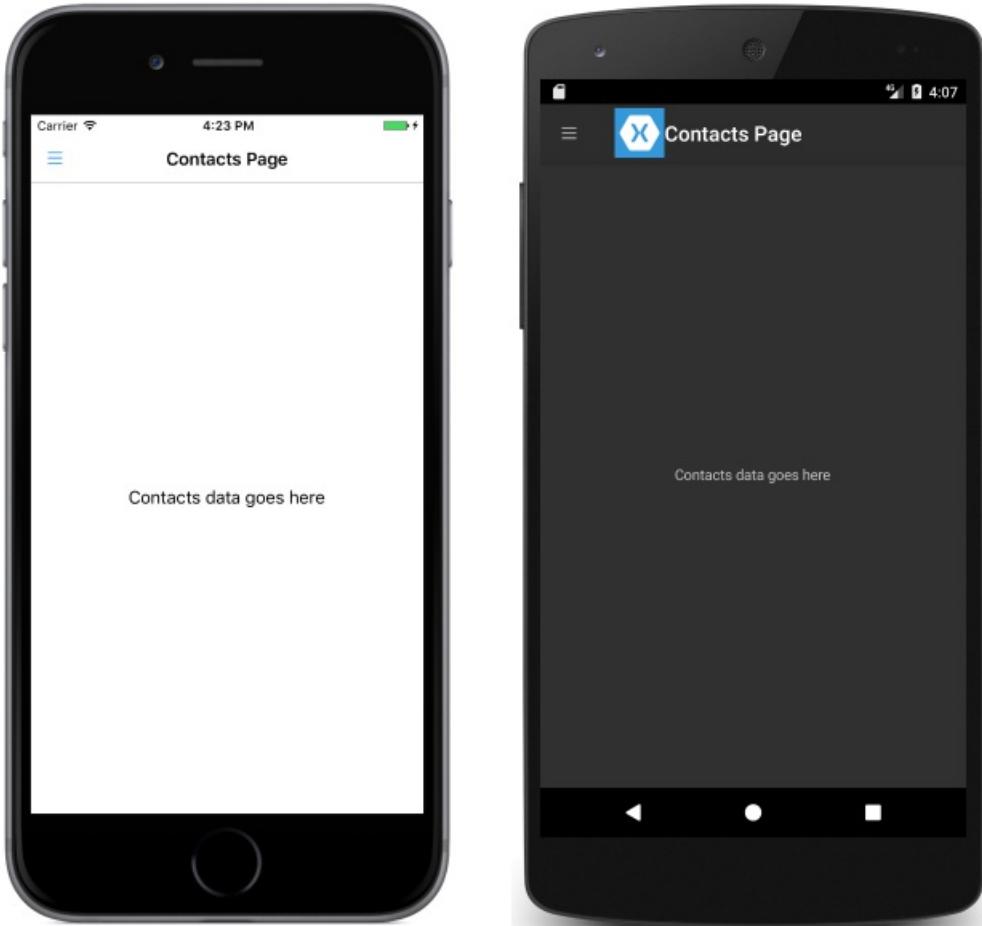
    void OnItemSelected (object sender, SelectedItemChangedEventArgs e)
    {
        var item = e.SelectedItem as MasterPageItem;
        if (item != null) {
            Detail = new NavigationPage ((Page)Activator.CreateInstance (item.TargetType));
            masterPage.listView.SelectedItem = null;
            IsPresented = false;
        }
    }
}
```

The `OnItemSelected` method performs the following actions:

- It retrieves the `SelectedItem` from the `ListView` instance, and provided that it's not `null`, sets the detail page to a new instance of the page type stored in the `TargetType` property of the `MasterPageItem`. The page type is wrapped in a `NavigationPage` instance to ensure that the icon referenced through the `Icon` property on the `MasterPage` is shown on the detail page in iOS.

- The selected item in the `ListView` is set to `null` to ensure that none of the `ListView` items will be selected next time the `MasterPage` is presented.
- The detail page is presented to the user by setting the `MasterDetailPage.IsPresented` property to `false`. This property controls whether the master or detail page is presented. It should be set to `true` to display the master page, and to `false` to display the detail page.

The following screenshots show the `ContactPage` detail page, which is shown after it's been selected on the master page:



### Controlling the Detail Page Display Behavior

How the `MasterDetailPage` manages the master and detail pages depends on whether the application is running on a phone or tablet, the orientation of the device, and the value of the `MasterBehavior` property. This property determines how the detail page will be displayed. Its possible values are:

- Default** – The pages are displayed using the platform default.
- Popover** – The detail page covers, or partially covers the master page.
- Split** – The master page is displayed on the left and the detail page is on the right.
- SplitOnLandscape** – A split screen is used when the device is in landscape orientation.
- SplitOnPortrait** – A split screen is used when the device is in portrait orientation.

The following XAML code example demonstrates how to set the `MasterBehavior` property on a `MasterDetailPage`:

```
<?xml version="1.0" encoding="UTF-8"?>
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MasterDetailPageNavigation.MainPage"
    MasterBehavior="Popover">
    ...
</MasterDetailPage>
```

The following code example shows the equivalent [MasterDetailPage](#) created in C#:

```
public class MainPageCS : MasterDetailPage
{
    MasterPageCS masterPage;

    public MainPageCS ()
    {
        MasterBehavior = MasterBehavior.Popover;
        ...
    }
}
```

However, the value of the [MasterBehavior](#) property only affects applications running on tablets or the desktop. Applications running on phones always have the *Popover* behavior.

## Summary

This article demonstrated how to use a [MasterDetailPage](#) and navigate between its pages of information. The Xamarin.Forms [MasterDetailPage](#) is a page that manages two pages of related information – a master page that presents items, and a detail page that presents details about items on the master page.

## Related Links

- [Page Varieties](#)
- [MasterDetailPage \(sample\)](#)
- [MasterDetailPage](#)

# Xamarin.Forms Modal Pages

7/12/2018 • 6 minutes to read • [Edit Online](#)

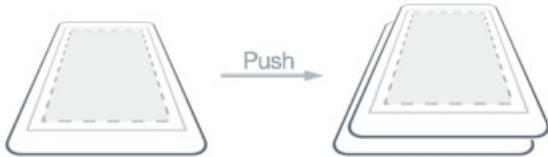
Xamarin.Forms provides support for modal pages. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled. This article demonstrates how to navigate to modal pages.

This article discusses the following topics:

- **Performing navigation** – pushing pages to the modal stack, popping pages from the modal stack, disabling the back button, and animating page transitions.
- **Passing data when navigating** – passing data through a page constructor, and through a `BindingContext`.

## Overview

A modal page can be any of the [Page](#) types supported by Xamarin.Forms. To display a modal page the application will push it onto the modal stack, where it will become the active page, as shown in the following diagram:



To return to the previous page the application will pop the current page from the modal stack, and the new topmost page becomes the active page, as shown in the following diagram:



## Performing Navigation

Modal navigation methods are exposed by the `Navigation` property on any [Page](#) derived types. These methods provide the ability to [push modal pages](#) onto the modal stack, and [pop modal pages](#) from the modal stack.

The `Navigation` property also exposes a `ModalStack` property from which the modal pages in the modal stack can be obtained. However, there is no concept of performing modal stack manipulation, or popping to the root page in modal navigation. This is because these operations are not universally supported on the underlying platforms.

### NOTE

A `NavigationPage` instance is not required for performing modal page navigation.

### Pushing Pages to the Modal Stack

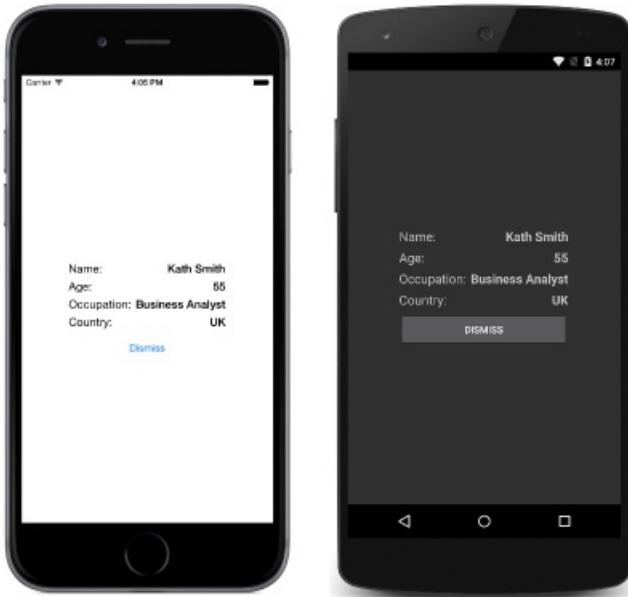
To navigate to the `ModalPage` it is necessary to invoke the `PushModalAsync` method on the `Navigation` property of the current page, as demonstrated in the following code example:

```

async void OnItemSelected (object sender, SelectedItemChangedEventArgs e)
{
    if (listView.SelectedItem != null) {
        var detailPage = new DetailPage ();
        ...
        await Navigation.PushModalAsync (detailPage);
    }
}

```

This causes the `ModalPage` instance to be pushed onto the modal stack, where it becomes the active page, provided that an item has been selected in the `ListView` on the `MainPage` instance. The `ModalPage` instance is shown in the following screenshots:



When `PushModalAsync` is invoked, the following events occur:

- The page calling `PushModalAsync` has its `OnDisappearing` override invoked, provided that the underlying platform isn't Android.
- The page being navigated to has its `OnAppearing` override invoked.
- The `PushAsync` task completes.

However, the precise order that these events occur is platform dependent. For more information, see [Chapter 24](#) of Charles Petzold's *Xamarin.Forms* book.

#### NOTE

Calls to the `OnDisappearing` and `OnAppearing` overrides cannot be treated as guaranteed indications of page navigation. For example, on iOS, the `onDisappearing` override is called on the active page when the application terminates.

### Popping Pages from the Modal Stack

The active page can be popped from the modal stack by pressing the *Back* button on the device, regardless of whether this is a physical button on the device or an on-screen button.

To programmatically return to the original page, the `ModalPage` instance must invoke the `PopModalAsync` method, as demonstrated in the following code example:

```
async void OnDismissButtonClicked (object sender, EventArgs args)
{
    await Navigation.PopModalAsync ();
}
```

This causes the `ModalPage` instance to be removed from the modal stack, with the new topmost page becoming the active page. When `PopModalAsync` is invoked, the following events occur:

- The page calling `PopModalAsync` has its `OnDisappearing` override invoked.
- The page being returned to has its `OnAppearing` override invoked, provided that the underlying platform isn't Android.
- The `PopModalAsync` task returns.

However, the precise order that these events occur is platform dependent. For more information, see [Chapter 24](#) of Charles Petzold's Xamarin.Forms book.

### Disabling the Back Button

On Android, the user can always return to the previous page by pressing the standard *Back* button on the device. If the modal page requires the user to complete a self-contained task before leaving the page, the application must disable the *Back* button. This can be accomplished by overriding the `Page.OnBackPressed` method on the modal page. For more information see [Chapter 24](#) of Charles Petzold's Xamarin.Forms book.

### Animating Page Transitions

The `Navigation` property of each page also provides overridden push and pop methods that include a `boolean` parameter that controls whether to display a page animation during navigation, as shown in the following code example:

```
async void OnNextPageButtonClicked (object sender, EventArgs e)
{
    // Page appearance not animated
    await Navigation.PushModalAsync (new DetailPage (), false);
}

async void OnDismissButtonClicked (object sender, EventArgs args)
{
    // Page appearance not animated
    await Navigation.PopModalAsync (false);
}
```

Setting the `boolean` parameter to `false` disables the page-transition animation, while setting the parameter to `true` enables the page-transition animation, provided that it is supported by the underlying platform. However, the push and pop methods that lack this parameter enable the animation by default.

## Passing Data when Navigating

Sometimes it's necessary for a page to pass data to another page during navigation. Two techniques for accomplishing this are by passing data through a page constructor, and by setting the new page's `BindingContext` to the data. Each will now be discussed in turn.

### Passing Data through a Page Constructor

The simplest technique for passing data to another page during navigation is through a page constructor parameter, which is shown in the following code example:

```
public App ()  
{  
    MainPage = new MainPage (DateTime.Now.ToString ("u"));  
}
```

This code creates a `MainPage` instance, passing in the current date and time in ISO8601 format.

The `MainPage` instance receives the data through a constructor parameter, as shown in the following code example:

```
public MainPage (string date)  
{  
    InitializeComponent ();  
    dateLabel.Text = date;  
}
```

The data is then displayed on the page by setting the `Label.Text` property.

### Passing Data through a BindingContext

An alternative approach for passing data to another page during navigation is by setting the new page's `BindingContext` to the data, as shown in the following code example:

```
async void OnItemSelected (object sender, SelectedItemChangedEventArgs e)  
{  
    if (listView.SelectedItem != null) {  
        var detailPage = new DetailPage ();  
        detailPage.BindingContext = e.SelectedItem as Contact;  
        listView.SelectedItem = null;  
        await Navigation.PushModalAsync (detailPage);  
    }  
}
```

This code sets the `BindingContext` of the `DetailPage` instance to the `Contact` instance, and then navigates to the `DetailPage`.

The `DetailPage` then uses data binding to display the `Contact` instance data, as shown in the following XAML code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"  
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"  
    x:Class="ModalNavigation.DetailPage">  
    <ContentPage.Padding>  
        <OnPlatform x:TypeArguments="Thickness">  
            <On Platform="iOS" Value="0,40,0,0" />  
        </OnPlatform>  
    </ContentPage.Padding>  
    <ContentPage.Content>  
        <StackLayout HorizontalOptions="Center" VerticalOptions="Center">  
            <StackLayout Orientation="Horizontal">  
                <Label Text="Name:" FontSize="Medium" HorizontalOptions="FillAndExpand" />  
                <Label Text="{Binding Name}" FontSize="Medium" FontAttributes="Bold" />  
            </StackLayout>  
            ...  
            <Button x:Name="dismissButton" Text="Dismiss" Clicked="OnDismissButtonClicked" />  
        </StackLayout>  
    </ContentPage.Content>  
</ContentPage>
```

The following code example shows how the data binding can be accomplished in C#:

```

public class DetailPageCS : ContentPage
{
    public DetailPageCS ()
    {
        var nameLabel = new Label {
            FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
            FontAttributes = FontAttributes.Bold
        };
        nameLabel.SetBinding (Label.TextProperty, "Name");
        ...
        var dismissButton = new Button { Text = "Dismiss" };
        dismissButton.Clicked += OnDismissButtonClicked;

        Thickness padding;
        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
                padding = new Thickness(0, 40, 0, 0);
                break;
            default:
                padding = new Thickness();
                break;
        }

        Padding = padding;
        Content = new StackLayout {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Children = {
                new StackLayout {
                    Orientation = StackOrientation.Horizontal,
                    Children = {
                        new Label{ Text = "Name:", FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                        HorizontalOptions = LayoutOptions.FillAndExpand },
                        nameLabel
                    }
                },
                ...
                dismissButton
            }
        };
    }

    async void OnDismissButtonClicked (object sender, EventArgs args)
    {
        await Navigation.PopModalAsync ();
    }
}

```

The data is then displayed on the page by a series of [Label](#) controls.

For more information about data binding, see [Data Binding Basics](#).

## Summary

This article demonstrated how to navigate to modal pages. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled.

## Related Links

- [Page Navigation](#)
- [Modal \(sample\)](#)
- [PassingData \(sample\)](#)



# Displaying Pop-ups

7/12/2018 • 2 minutes to read • [Edit Online](#)

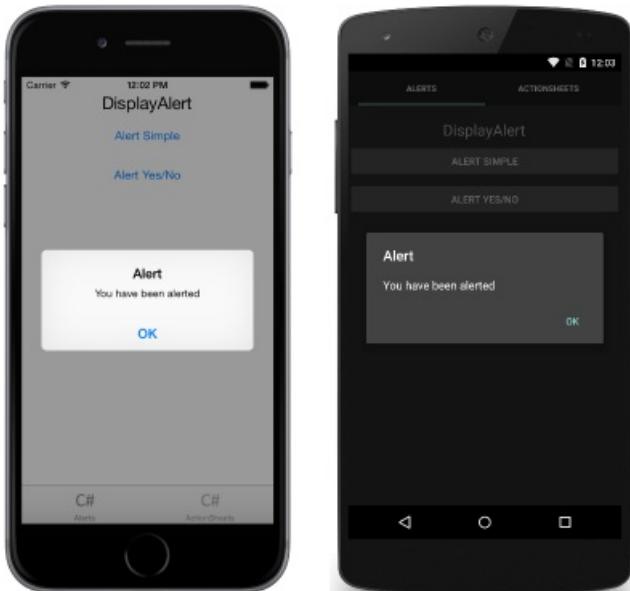
Xamarin.Forms provides two pop-up-like user interface elements – an alert and an action sheet. This article demonstrates using the alert and action sheet APIs to ask users simple questions and to guide users through tasks.

Displaying an alert or asking a user to make a choice is a common UI task. Xamarin.Forms has two methods on the `Page` class for interacting with the user via a pop-up: `DisplayAlert` and `DisplayActionSheet`. They are rendered with appropriate native controls on each platform.

## Displaying an Alert

All Xamarin.Forms-supported platforms have a modal pop-up to alert the user or ask simple questions of them. To display these alerts in Xamarin.Forms, use the `DisplayAlert` method on any `Page`. The following line of code shows a simple message to the user:

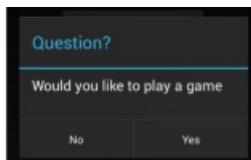
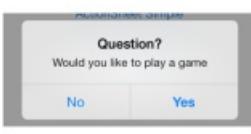
```
DisplayAlert ("Alert", "You have been alerted", "OK");
```



This example does not collect information from the user. The alert displays modally and once dismissed the user continues interacting with the application.

The `DisplayAlert` method can also be used to capture a user's response by presenting two buttons and returning a `boolean`. To get a response from an alert, supply text for both buttons and `await` the method. After the user selects one of the options the answer will be returned to your code. Note the `async` and `await` keywords in the sample code below:

```
async void OnAlertYesNoClicked (object sender, EventArgs e)
{
    var answer = await DisplayAlert ("Question?", "Would you like to play a game", "Yes", "No");
    Debug.WriteLine ("Answer: " + answer);
}
```

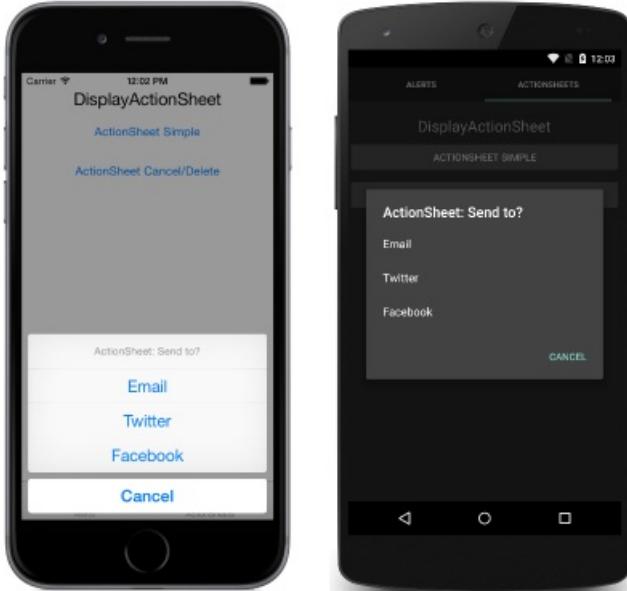


## Guiding Users Through Tasks

The [UIAlertAction](#) is a common UI element in iOS. The `Xamarin.Forms` [DisplayActionSheet](#) method lets you include this control in cross-platform apps, rendering native alternatives in Android and UWP.

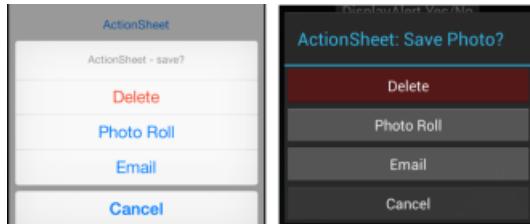
To display an action sheet, `await DisplayActionSheet` in any [Page](#), passing the message and button labels as strings. The method returns the string label of the button that was clicked by the user. A simple example is shown here:

```
async void OnActionSheetSimpleClicked (object sender, EventArgs e)
{
    var action = await DisplayActionSheet ("ActionSheet: Send to?", "Cancel", null, "Email", "Twitter",
    "Facebook");
    Debug.WriteLine ("Action: " + action);
}
```



The `destroy` button is rendered differently than the others, and can be left `null` or specified as the third string parameter. The following example uses the `destroy` button:

```
async void OnActionSheetCancelDeleteClicked (object sender, EventArgs e)
{
    var action = await DisplayActionSheet ("ActionSheet: SavePhoto?", "Cancel", "Delete", "Photo Roll",
    "Email");
    Debug.WriteLine ("Action: " + action);
}
```



## Summary

This article demonstrated using the alert and action sheet APIs to ask users simple questions and to guide users through tasks. Xamarin.Forms has two methods on the `Page` class for interacting with the user via a pop-up: `DisplayAlert` and `DisplayActionSheet`, and they are both rendered with appropriate native controls on each platform.

## Related Links

- [PopupsSample](#)

# Xamarin.Forms Templates

7/12/2018 • 2 minutes to read • [Edit Online](#)

## Control Templates

Xamarin.Forms control templates provide the ability to easily theme and re-theme application pages at runtime.

## Data Templates

Xamarin.Forms data templates provide the ability to define the presentation of data on supported controls.

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms Gallery \(sample\)](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms Control Templates

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Control templates provide a clean separation between the appearance of a page and its content, enabling the creation of pages that can easily be themed.*

## Introduction

Xamarin.Forms control templates provide the ability to easily theme and re-theme application pages at runtime. This article provides an introduction to control templates.

## Creating a ControlTemplate

Control templates can be defined at the application level or at the page level. This article demonstrates how to create and consume control templates.

## Binding from a ControlTemplate

Template bindings allow controls in a control template to data bind to public properties, enabling property values on controls in the control template to be easily changed. This article demonstrates using template bindings to perform data binding from a control template.

# Introduction to Xamarin.Forms Control Templates

7/12/2018 • 3 minutes to read • [Edit Online](#)

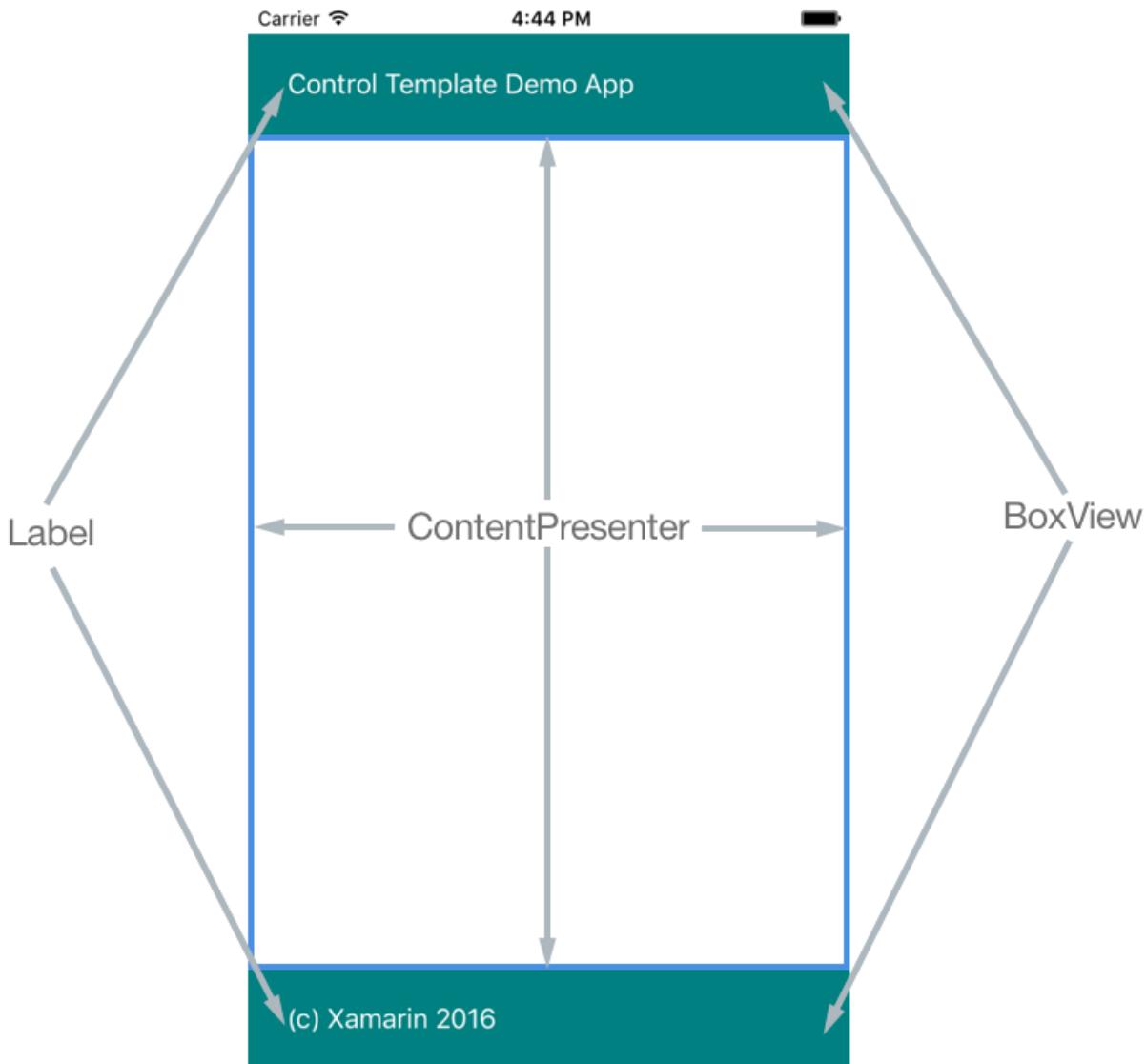
Xamarin.Forms control templates provide the ability to easily theme and re-theme application pages at runtime. This article provides an introduction to control templates.

Controls have different properties, such as `BackgroundColor` and `TextColor`, that can define aspects of the control's appearance. These properties can be set using [styles](#), which can be changed at runtime to implement basic theming. However, styles don't maintain a clean separation between the appearance of a page and its content, and the changes that can be made by setting such properties are limited.

Control templates provide a clean separation between the appearance of a page and its content, therefore enabling the creation of pages that can easily be themed. For example, an application may contain application-level control templates that provide a dark theme and a light theme. Each `ContentPage` in the application can be themed by applying one of the control templates without changing the content being displayed by each page. In addition, the themes provided by control templates aren't limited to changing the properties of controls. They can also change the controls used to implement the theme.

## Creating a ControlTemplate

A `ControlTemplate` specifies the appearance of a page or view, and contains a root layout, and within the layout, the controls that implement the template. Typically, a `ControlTemplate` will utilize a `ContentPresenter` to mark where the content to be displayed by the page or view will appear. The page or view that consumes the `ControlTemplate` will then define content to be displayed by the `ContentPresenter`. The following diagram illustrates a `ControlTemplate` for a page that contains a number of controls, including a `ContentPresenter` marked by a blue rectangle:



A `ControlTemplate` can be applied to the following types by setting their `ControlTemplate` properties:

- `ContentPage`
- `ContentView`
- `TemplatedPage`
- `TemplatedView`

When a `ControlTemplate` is created and assigned to these types, any existing appearance is replaced with the appearance defined in the `ControlTemplate`. In addition, as well as setting appearance by using the `ControlTemplate` property, control templates can also be applied by using styles to further expand theme ability.

#### NOTE

What are the `TemplatedPage` and `TemplatedView` types? `TemplatedPage` is the base class for `ContentPage`, and is the most basic page type provided by Xamarin.Forms. Unlike `ContentPage`, `TemplatedPage` does not have a `Content` property. Therefore, content can't be directly added to a `TemplatedPage` instance. Instead, content is added by setting the control template for the `TemplatedPage` instance. Similarly, `TemplatedView` is the base class for `ContentView`. Unlike `ContentView`, `TemplatedView` does not have a `Content` property. Therefore, content can't be directly added to a `TemplatedView` instance. Instead, content is added by setting the control template for the `TemplatedView` instance.

Control templates can be created in XAML and in C#:

- Control templates created in XAML are defined in a `ResourceDictionary` that's assigned to the `Resources` collection of a page, or more typically to the `Resources` collection of the application.

- Control templates created in C# are typically defined in the page's class, or in a class that can be globally accessed.

Choosing where to define a `ControlTemplate` instance impacts where it can be used:

- `ControlTemplate` instances defined at the page-level can only be applied to the page.
- `ControlTemplate` instances defined at the application-level can be applied to pages throughout the application.

Control templates lower in the view hierarchy take precedence over those defined higher up. For example, a `ControlTemplate` named `DarkTheme` that's defined at the page-level will take precedence over an identically named template defined at the application-level. Therefore, a control template that defines a theme to be applied to each page in an application should be defined at the application-level.

## Related Links

- [Styles](#)
- [ControlTemplate](#)
- [ContentPresenter](#)

# Creating a ControlTemplate

7/12/2018 • 4 minutes to read • [Edit Online](#)

*Control templates can be defined at the application level or page level. This article demonstrates how to create and consume control templates.*

## Creating a ControlTemplate in XAML

To define a `ControlTemplate` at the application level, a `ResourceDictionary` must be added to the `App` class. By default, all Xamarin.Forms applications created from a template use the `App` class to implement the `Application` subclass. To declare a `ControlTemplate` at the application level, in the application's `ResourceDictionary` using XAML, the default `App` class must be replaced with a XAML `App` class and associated code-behind, as shown in the following code example:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.App">
    <Application.Resources>
        <ResourceDictionary>
            <ControlTemplate x:Key="TealTemplate">
                <Grid>
                    ...
                    <BoxView ... />
                    <Label Text="Control Template Demo App"
                        TextColor="White"
                        VerticalOptions="Center" ... />
                    <ContentPresenter ... />
                    <BoxView Color="Teal" ... />
                    <Label Text="(c) Xamarin 2016"
                        TextColor="White"
                        VerticalOptions="Center" ... />
                </Grid>
            </ControlTemplate>
            <ControlTemplate x:Key="AquaTemplate">
                ...
            </ControlTemplate>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Each `ControlTemplate` instance is created as a reusable object in a `ResourceDictionary`. This is achieved by giving each declaration a unique `x:Key` attribute, which provides it with a descriptive key in the `ResourceDictionary`.

The following code example shows the associated `App` code-behind:

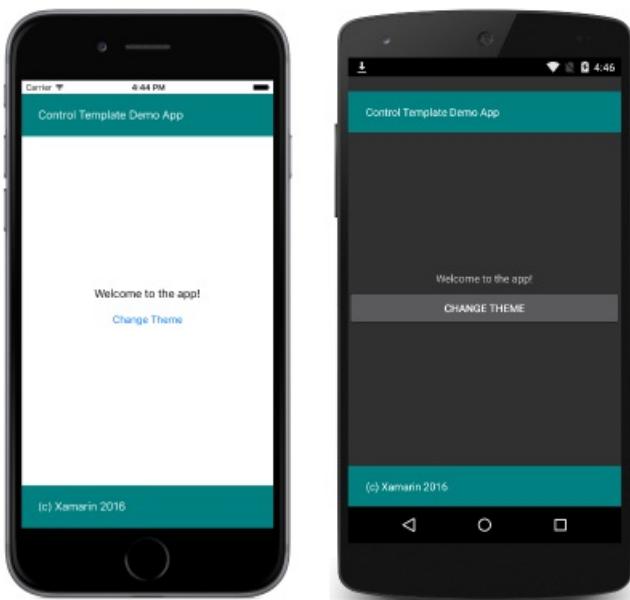
```
public partial class App : Application
{
    public App ()
    {
        InitializeComponent ();
        MainPage = new HomePage ();
    }
}
```

As well as setting the `MainPage` property, the code-behind must also call the `InitializeComponent` method to load and parse the associated XAML.

The following code example shows a `ContentPage` applying the `TealTemplate` to the `ContentView`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentView x:Name="contentView" Padding="0,20,0,0"
        ControlTemplate="{StaticResource TealTemplate}">
        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Welcome to the app!" HorizontalOptions="Center" />
            <Button Text="Change Theme" Clicked="OnButtonClicked" />
        </StackLayout>
    </ContentView>
</ContentPage>
```

The `TealTemplate` is assigned to the `ContentView.ControlTemplate` property by using the `StaticResource` markup extension. The `ContentView.Content` property is set to a `StackLayout` that defines the content to be displayed on the `ContentPage`. This content will be displayed by the `ContentPresenter` contained in the `TealTemplate`. This results in the appearance shown in the following screenshots:



### Re-theming an Application at Runtime

Clicking the **Change Theme** button executes the `OnButtonClicked` method, which is shown in the following code example:

```
void OnButtonClicked (object sender, EventArgs e)
{
    originalTemplate = !originalTemplate;
    contentView.ControlTemplate = (originalTemplate) ? tealTemplate : aquaTemplate;
}
```

This method replaces the active `ControlTemplate` instance with the alternative `ControlTemplate` instance, resulting in the following screenshot:



#### NOTE

On a `ContentPage`, the `Content` property can be assigned and the `ControlTemplate` property can also be set. When this occurs, if the `ControlTemplate` contains a `ContentPresenter` instance, the content assigned to the `content` property will be presented by the `ContentPresenter` within the `ControlTemplate`.

### Setting a ControlTemplate with a Style

A `ControlTemplate` can also be applied via a `Style` to further expand theme ability. This can be achieved by creating an *implicit* or *explicit* style for the target view in a `ResourceDictionary`, and setting the `ControlTemplate` property of the target view in the `Style` instance. The following code example shows an *implicit* style that's been added to the application level `ResourceDictionary`:

```
<Style TargetType="ContentView">
    <Setter Property="ControlTemplate" Value="{StaticResource TealTemplate}" />
</Style>
```

Because the `Style` instance is *implicit*, it will be applied to all `ContentView` instances in the application. Therefore, it's no longer necessary to set the `ContentView.ControlTemplate` property, as demonstrated in the following code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentView x:Name="contentView" Padding="0,20,0,0">
        ...
    </ContentView>
</ContentPage>
```

For more information about styles, see [Styles](#).

### Creating a ControlTemplate at Page Level

In addition to creating `ControlTemplate` instances at the application level, they can also be created at the page level, as shown in the following code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentPage.Resources>
        <ResourceDictionary>
            <ControlTemplate x:Key="TealTemplate">
                ...
            </ControlTemplate>
            <ControlTemplate x:Key="AquaTemplate">
                ...
            </ControlTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentView ... ControlTemplate="{StaticResource TealTemplate}">
        ...
    </ContentView>
</ContentPage>

```

When adding a `ControlTemplate` at the page level, a `ResourceDictionary` is added to the `ContentPage`, and then the `ControlTemplate` instances are included in the `ResourceDictionary`.

## Creating a ControlTemplate in C#

To define a `ControlTemplate` at the application level, a `class` must be created that represents the `ControlTemplate`. The class should derive from the `layout` being used for the template, as shown in the following code example:

```

class TealTemplate : Grid
{
    public TealTemplate ()
    {
        ...
        var contentPresenter = new ContentPresenter ();
        Children.Add (contentPresenter, 0, 1);
        Grid.SetColumnSpan (contentPresenter, 2);
        ...
    }
}

class AquaTemplate : Grid
{
    ...
}

```

The `AquaTemplate` class is identical to the `TealTemplate` class, except that different colors are used for the `BoxView.Color` and `Label.TextColor` properties.

The following code example shows a `ContentPage` applying the `TealTemplate` to the `ContentView`:

```

public class HomePageCS : ContentPage
{
    ...
    ControlTemplate tealTemplate = new ControlTemplate (typeof(TealTemplate));
    ControlTemplate aquaTemplate = new ControlTemplate (typeof(AquaTemplate));

    public HomePageCS ()
    {
        var button = new Button { Text = "Change Theme" };
        var contentView = new ContentView {
            Padding = new Thickness (0, 20, 0, 0),
            Content = new StackLayout {
                VerticalOptions = LayoutOptions.CenterAndExpand,
                Children = {
                    new Label { Text = "Welcome to the app!", HorizontalOptions = LayoutOptions.Center },
                    button
                }
            },
            ControlTemplate = tealTemplate
        };
        ...
        Content = contentView;
    }
}

```

The `ControlTemplate` instances are created by specifying the type of the classes that define the control templates, in the `ControlTemplate` constructor.

The `ContentView.Content` property is set to a `StackLayout` that defines the content to be displayed on the `ContentPage`. This content will be displayed by the `ContentPresenter` contained in the `TealTemplate`. The same mechanism outlined previously is used to change the theme at runtime to the `AquaTheme`.

## Summary

This article demonstrated how to create and consume control templates. Control templates can be defined at the application level or page level.

## Related Links

- [Styles](#)
- [Simple Theme \(sample\)](#)
- [ControlTemplate](#)
- [ContentPresenter](#)
- [ContentView](#)
- [ResourceDictionary](#)

# Binding from a Xamarin.Forms ControlTemplate

10/23/2018 • 4 minutes to read • [Edit Online](#)

Template bindings allow controls in a control template to data bind to public properties, enabling property values on controls in the control template to be easily changed. This article demonstrates using template bindings to perform data binding from a control template.

A `TemplateBinding` is used to bind a control's property in a control template to a bindable property on the parent of the *target* view that owns the control template. For example, rather than defining the text displayed by `Label` instances inside the `ControlTemplate`, you could use a template binding to bind the `Label.Text` property to bindable properties that define the text to be displayed.

A `TemplateBinding` is similar to an existing `Binding`, except that the *source* of a `TemplateBinding` is always automatically set to the parent of the *target* view that owns the control template. However, note that using a `TemplateBinding` outside of a `ControlTemplate` is not supported.

## Creating a TemplateBinding in XAML

In XAML, a `TemplateBinding` is created using the `TemplateBinding` markup extension, as demonstrated in the following code example:

```
<ControlTemplate x:Key="TealTemplate">
    <Grid>
        ...
        <Label Text="{TemplateBinding Parent.HeaderText}" ... />
        ...
        <Label Text="{TemplateBinding Parent.FooterText}" ... />
    </Grid>
</ControlTemplate>
```

Rather than set the `Label.Text` properties to static text, the properties can use template bindings to bind to bindable properties on the parent of the *target* view that owns the `ControlTemplate`. However, note that the template bindings bind to `Parent.HeaderText` and `Parent.FooterText`, rather than `HeaderText` and `FooterText`. This is because in this example, the bindable properties are defined on the grandparent of the *target* view, rather than the parent, as demonstrated in the following code example:

```
<ContentPage ...>
    <ContentView ... ControlTemplate="{StaticResource TealTemplate}">
        ...
    </ContentView>
</ContentPage>
```

The *source* of the template binding is always automatically set to the parent of the *target* view that owns the control template, which here is the `ContentView` instance. The template binding uses the `Parent` property to return the parent element of the `ContentView` instance, which is the `ContentPage` instance. Therefore, using a `TemplateBinding` in the `ControlTemplate` to bind to `Parent.HeaderText` and `Parent.FooterText` locates the bindable properties that are defined on the `ContentPage`, as demonstrated in the following code example:

```

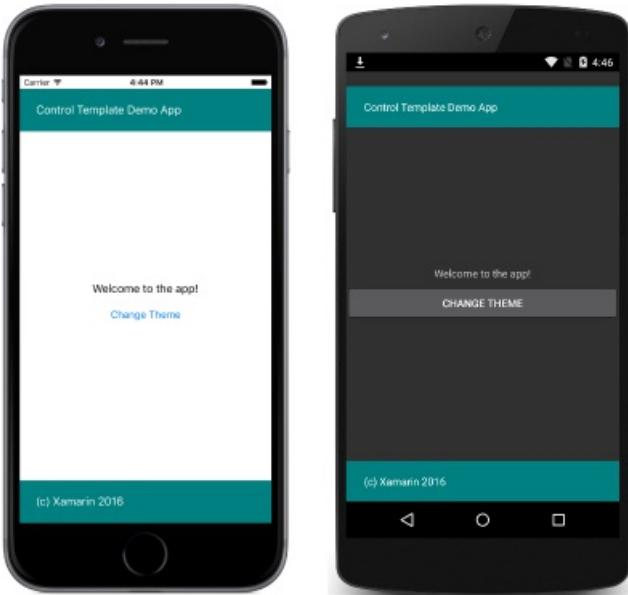
public static readonly BindableProperty HeaderTextProperty =
    BindableProperty.Create ("HeaderText", typeof(string), typeof(HomePage), "Control Template Demo App");
public static readonly BindableProperty FooterTextProperty =
    BindableProperty.Create ("FooterText", typeof(string), typeof(HomePage), "(c) Xamarin 2016");

public string HeaderText {
    get { return (string)GetValue (HeaderTextProperty); }
}

public string FooterText {
    get { return (string)GetValue (FooterTextProperty); }
}

```

This results in the appearance shown in the following screenshots:



## Creating a TemplateBinding in C#

In C#, a `TemplateBinding` is created by using the `TemplateBinding` constructor, as demonstrated in the following code example:

```

class TealTemplate : Grid
{
    public TealTemplate ()
    {
        ...
        var topLabel = new Label { TextColor = Color.White, VerticalOptions = LayoutOptions.Center };
        topLabel.SetBinding (Label.TextProperty, new TemplateBinding ("Parent.HeaderText"));
        ...
        var bottomLabel = new Label { TextColor = Color.White, VerticalOptions = LayoutOptions.Center };
        bottomLabel.SetBinding (Label.TextProperty, new TemplateBinding ("Parent.FooterText"));
        ...
    }
}

```

Rather than set the `Label.Text` properties to static text, the properties can use template bindings to bind to bindable properties on the parent of the *target* view that owns the `ControlTemplate`. The template binding is created by using the `SetBinding` method, specifying a `TemplateBinding` instance as the second parameter. Note that the template bindings bind to `Parent.HeaderText` and `Parent.FooterText`, because the bindable properties are defined on the grandparent of the *target* view, rather than the parent, as demonstrated in the following code example:

```

public class HomePageCS : ContentPage
{
    ...
    public HomePageCS ()
    {
        Content = new ContentView {
            ControlTemplate = tealTemplate,
            Content = new StackLayout {
                ...
            },
            ...
        };
        ...
    }
}

```

The bindable properties are defined on the `ContentPage`, as outlined earlier.

### Binding a BindableProperty to a ViewModel Property

As previously stated, a `TemplateBinding` binds a control's property in a control template to a bindable property on the parent of the *target* view that owns the control template. In turn, these bindable properties can be bound to properties in ViewModels.

The following code example defines two properties on a ViewModel:

```

public class HomePageViewModel
{
    public string HeaderText { get { return "Control Template Demo App"; } }
    public string FooterText { get { return "(c) Xamarin 2016"; } }
}

```

The `HeaderText` and `FooterText` ViewModel properties can be bound to, as shown in the following XAML code example:

```

<ContentPage xmlns:local="clr-namespace:SimpleTheme;assembly=SimpleTheme"
    HeaderText="{Binding HeaderText}" FooterText="{Binding FooterText}" ...>
    <ContentPage.BindingContext>
        <local:HomePageViewModel />
    </ContentPage.BindingContext>
    <ContentView ControlTemplate="{StaticResource TealTemplate}" ...>
        ...
    </ContentView>
</ContentPage>

```

The `HeaderText` and `FooterText` bindable properties are bound to the `HomePageViewModel.HeaderText` and `HomePageViewModel.FooterText` properties, due to setting the `BindingContext` to an instance of the `HomePageViewModel` class. Overall, this results in control properties in the `ControlTemplate` being bound to `BindableProperty` instances on the `ContentPage`, which in turn bind to ViewModel properties.

The equivalent C# code is shown in the following code example:

```
public class HomePageCS : ContentPage
{
    ...
    public HomePageCS ()
    {
        BindingContext = new HomePageViewModel ();
        this.SetBinding (HeaderTextProperty, "HeaderText");
        this.SetBinding (FooterTextProperty, "FooterText");
        ...
    }
}
```

You can also bind to the view model properties directly, so that you don't need to declare `BindableProperty`s for `HeaderText` and `FooterText` on the `ContentPage`, by binding the control template to `Parent.BindingContext.PropertyName` e.g.:

```
<ControlTemplate x:Key="TealTemplate">
<Grid>
    ...
    <Label Text="{TemplateBinding Parent.BindingContext.HeaderText}" ... />
    ...
    <Label Text="{TemplateBinding Parent.BindingContext.FooterText}" ... />
</Grid>
</ControlTemplate>
```

For more information about data binding to ViewModels, see [From Data Bindings to MVVM](#).

## Summary

This article demonstrated using template bindings to perform data binding from a control template. Template bindings allow controls in a control template to data bind to public properties, enabling property values on controls in the control template to be easily changed.

## Related Links

- [Data Binding Basics](#)
- [From Data Bindings to MVVM](#)
- [Simple Theme with Template Binding \(sample\)](#)
- [Simple Theme with Template Binding and ViewModel \(sample\)](#)
- [TemplateBinding](#)
- [ControlTemplate](#)
- [ContentView](#)

# Xamarin.Forms Data Templates

7/12/2018 • 2 minutes to read • [Edit Online](#)

A *DataTemplate* is used to specify the appearance of data on supported controls, and typically binds to the data to be displayed.

## Introduction

Xamarin.Forms data templates provide the ability to define the presentation of data on supported controls. This article provides an introduction to data templates, examining why they are necessary.

## Creating a DataTemplate

Data templates can be created inline, in a [ResourceDictionary](#), or from a custom type or appropriate Xamarin.Forms cell type. An inline template should be used if there's no need to reuse the data template elsewhere. Alternatively, a data template can be reused by defining it as a custom type, or as a control-level, page-level, or application-level resource.

## Creating a DataTemplateSelector

A [DataTemplateSelector](#) can be used to choose a [DataTemplate](#) at runtime based on the value of a data-bound property. This enables multiple [DataTemplate](#) instances to be applied to the same type of object, to customize the appearance of particular objects. This article demonstrates how to create and consume a [DataTemplateSelector](#).

## Related Links

- [Data Templates \(sample\)](#)

# Introduction to Xamarin.Forms Data Templates

7/12/2018 • 3 minutes to read • [Edit Online](#)

Xamarin.Forms data templates provide the ability to define the presentation of data on supported controls. This article provides an introduction to data templates, examining why they are necessary.

Consider a `ListView` that displays a collection of `Person` objects. The following code example shows the definition of the `Person` class:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Location { get; set; }
}
```

The `Person` class defines `Name`, `Age`, and `Location` properties, which can be set when a `Person` object is created. The `ListView` is used to display the collection of `Person` objects, as shown in the following XAML code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataTemplates"
    ...>
    <StackLayout Margin="20">
        ...
        <ListView Margin="0,20,0,0">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    <local:Person Name="John" Age="37" Location="USA" />
                    <local:Person Name="Tom" Age="42" Location="UK" />
                    <local:Person Name="Lucas" Age="29" Location="Germany" />
                    <local:Person Name="Tariq" Age="39" Location="UK" />
                    <local:Person Name="Jane" Age="30" Location="USA" />
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </StackLayout>
</ContentPage>
```

Items are added to the `ListView` in XAML by initializing the `ItemsSource` property from an array of `Person` instances.

## NOTE

Note that the `x:Array` element requires a `Type` attribute indicating the type of the items in the array.

The equivalent C# page is shown in the following code example, which initializes the `ItemsSource` property to a `List` of `Person` instances:

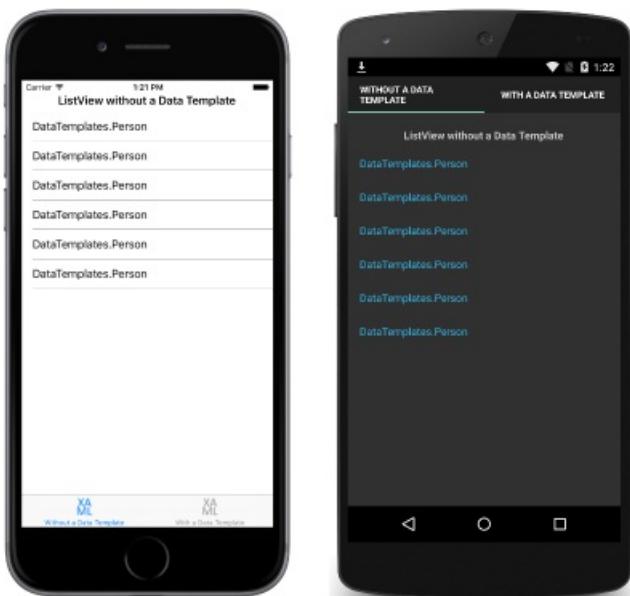
```

public WithoutDataTemplatePageCS()
{
    ...
    var people = new List<Person>
    {
        new Person { Name = "Steve", Age = 21, Location = "USA" },
        new Person { Name = "John", Age = 37, Location = "USA" },
        new Person { Name = "Tom", Age = 42, Location = "UK" },
        new Person { Name = "Lucas", Age = 29, Location = "Germany" },
        new Person { Name = "Tariq", Age = 39, Location = "UK" },
        new Person { Name = "Jane", Age = 30, Location = "USA" }
    };

    Content = new StackLayout
    {
        Margin = new Thickness(20),
        Children = {
            ...
            new ListView { ItemsSource = people, Margin = new Thickness(0, 20, 0, 0) }
        }
    };
}

```

The `ListView` calls `ToString` when displaying the objects in the collection. Because there is no `Person.ToString` override, `ToString` returns the type name of each object, as shown in the following screenshots:



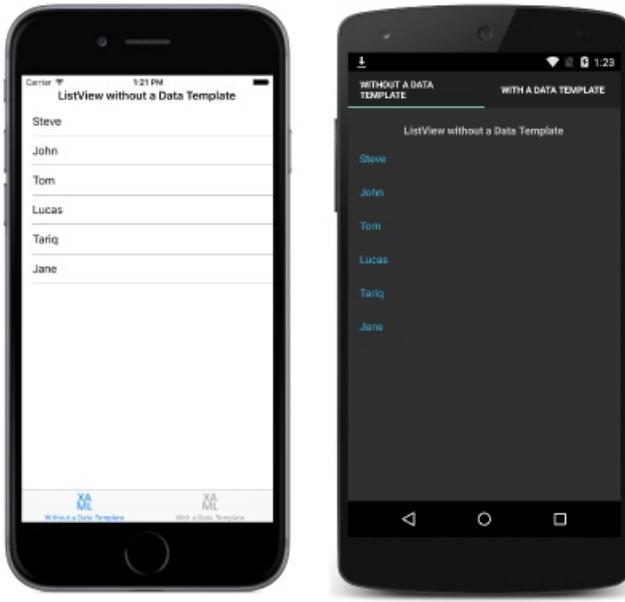
The `Person` object can override the `ToString` method to display meaningful data, as shown in the following code example:

```

public class Person
{
    ...
    public override string ToString ()
    {
        return Name;
    }
}

```

This results in the `ListView` displaying the `Person.Name` property value for each object in the collection, as shown in the following screenshots:



The `Person.ToString` override could return a formatted string consisting of the `Name`, `Age`, and `Location` properties. However, this approach offers only a limited control over the appearance of each item of data. For more flexibility, a `DataTemplate` can be created that defines the appearance of the data.

## Creating a DataTemplate

A `DataTemplate` is used to specify the appearance of data, and typically uses data binding to display data. Its common usage scenario is when displaying data from a collection of objects in a `ListView`. For example, when a `ListView` is bound to a collection of `Person` objects, the `ListView.ItemTemplate` property will be set to a `DataTemplate` that defines the appearance of each `Person` object in the `ListView`. The `DataTemplate` will contain elements that bind to property values of each `Person` object. For more information about data binding, see [Data Binding Basics](#).

A `DataTemplate` can be used as a value for the following properties:

- `ListView.HeaderTemplate`
- `ListView.FooterTemplate`
- `ListView.GroupHeaderTemplate`
- `ItemsView.ItemTemplate`, which is inherited by `ListView`.
- `MultiPage.ItemTemplate`, which is inherited by `CarouselPage`, `MasterDetailPage`, and `TabbedPage`.

### NOTE

Note that although the `TableView` makes use of `Cell` objects, it does not use a `DataTemplate`. This is because data bindings are always set directly on `cell` objects.

A `DataTemplate` that's placed as a direct child of the properties listed above is known as an *inline template*. Alternatively, a `DataTemplate` can be defined as a control-level, page-level, or application-level resource. Choosing where to define a `DataTemplate` impacts where it can be used:

- A `DataTemplate` defined at the control level can only be applied to the control.
- A `DataTemplate` defined at the page level can be applied to multiple valid controls on the page.
- A `DataTemplate` defined at the application level can be applied to valid controls throughout the application.

Data templates lower in the view hierarchy take precedence over those defined higher up when they share `x:Key` attributes. For example, an application-level data template will be overridden by a page-level data template, and a

page-level data template will be overridden by a control-level data template, or an inline data template.

## Related Links

- [Cell Appearance](#)
- [Data Templates \(sample\)](#)
- [DataTemplate](#)

# Creating a Xamarin.Forms DataTemplate

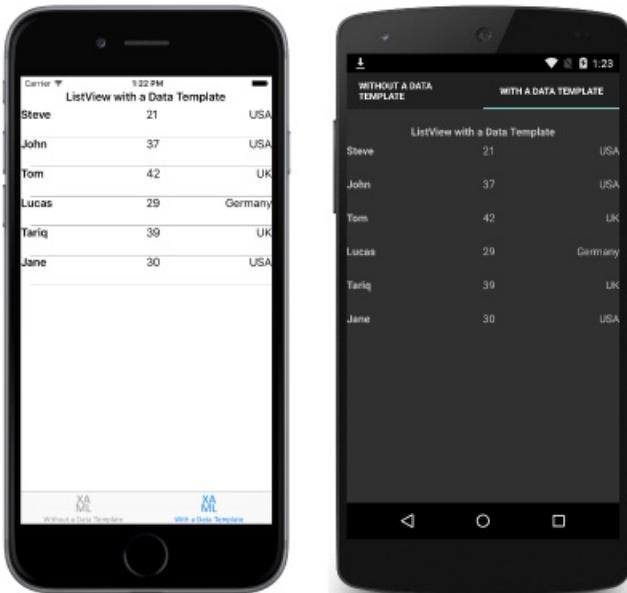
7/12/2018 • 5 minutes to read • [Edit Online](#)

Data templates can be created inline, in a ResourceDictionary, or from a custom type or appropriate Xamarin.Forms cell type. This article explores each technique.

A common usage scenario for a `DataTemplate` is displaying data from a collection of objects in a `ListView`. The appearance of the data for each cell in the `ListView` can be managed by setting the `ListView.ItemTemplate` property to a `DataTemplate`. There are a number of techniques that can be used to accomplish this:

- [Creating an Inline DataTemplate](#).
- [Creating a DataTemplate with a Type](#).
- [Creating a DataTemplate as a Resource](#).

Regardless of the technique being used, the result is that the appearance of each cell in the `ListView` is defined by a `DataTemplate`, as shown in the following screenshots:



## Creating an Inline DataTemplate

The `ListView.ItemTemplate` property can be set to an inline `DataTemplate`. An inline template, which is one that's placed as a direct child of an appropriate control property, should be used if there's no need to reuse the data template elsewhere. The elements specified in the `DataTemplate` define the appearance of each cell, as shown in the following XAML code example:

```

<ListView Margin="0,20,0,0">
    <ListView.ItemsSource>
        <x:Array Type="{x:Type local:Person}">
            <local:Person Name="Steve" Age="21" Location="USA" />
            <local:Person Name="John" Age="37" Location="USA" />
            <local:Person Name="Tom" Age="42" Location="UK" />
            <local:Person Name="Lucas" Age="29" Location="Germany" />
            <local:Person Name="Tariq" Age="39" Location="UK" />
            <local:Person Name="Jane" Age="30" Location="USA" />
        </x:Array>
    </ListView.ItemsSource>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <Grid>
                    ...
                    <Label Text="{Binding Name}" FontAttributes="Bold" />
                    <Label Grid.Column="1" Text="{Binding Age}" />
                    <Label Grid.Column="2" Text="{Binding Location}" HorizontalTextAlignment="End" />
                </Grid>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

The child of an inline `DataTemplate` must be of, or derive from, type `ViewCell`. Layout inside the `ViewCell` is managed here by a `Grid`. The `Grid` contains three `Label` instances that bind their `Text` properties to the appropriate properties of each `Person` object in the collection.

The equivalent C# code is shown in the following code example:

```

public class WithDataTemplatePageCS : ContentPage
{
    public WithDataTemplatePageCS()
    {
        ...
        var people = new List<Person>
        {
            new Person { Name = "Steve", Age = 21, Location = "USA" },
            ...
        };

        var personDataTemplate = new DataTemplate(() =>
        {
            var grid = new Grid();
            ...
            var nameLabel = new Label { FontAttributes = FontAttributes.Bold };
            var ageLabel = new Label();
            var locationLabel = new Label { HorizontalTextAlignment = TextAlignment.End };

            nameLabel.SetBinding(Label.TextProperty, "Name");
            ageLabel.SetBinding(Label.TextProperty, "Age");
            locationLabel.SetBinding(Label.TextProperty, "Location");

            grid.Children.Add(nameLabel);
            grid.Children.Add(ageLabel, 1, 0);
            grid.Children.Add(locationLabel, 2, 0);

            return new ViewCell { View = grid };
        });

        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children = {
                ...
                new ListView { ItemsSource = people, ItemTemplate = personDataTemplate, Margin = new Thickness(0, 20, 0, 0) }
            }
        };
    }
}

```

In C#, the inline `DataTemplate` is created using a constructor overload that specifies a `Func` argument.

## Creating a DataTemplate with a Type

The `ListView.ItemTemplate` property can also be set to a `DataTemplate` that's created from a cell type. The advantage of this approach is that the appearance defined by the cell type can be reused by multiple data templates throughout the application. The following XAML code shows an example of this approach:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataTemplates"
    ...>
    <StackLayout Margin="20">
        ...
        <ListView Margin="0,20,0,0">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    ...
                </x:Array>
            </ListView.ItemsSource>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <local:PersonCell />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>

```

Here, the `ListView.ItemTemplate` property is set to a `DataTemplate` that's created from a custom type that defines the cell appearance. The custom type must derive from type `ViewCell`, as shown in the following code example:

```

<ViewCell xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataTemplates.PersonCell">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.5*" />
            <ColumnDefinition Width="0.2*" />
            <ColumnDefinition Width="0.3*" />
        </Grid.ColumnDefinitions>
        <Label Text="{Binding Name}" FontAttributes="Bold" />
        <Label Grid.Column="1" Text="{Binding Age}" />
        <Label Grid.Column="2" Text="{Binding Location}" HorizontalTextAlignment="End" />
    </Grid>
</ViewCell>

```

Within the `ViewCell`, layout is managed here by a `Grid`. The `Grid` contains three `Label` instances that bind their `Text` properties to the appropriate properties of each `Person` object in the collection.

The equivalent C# code is shown in the following example:

```

public class WithDataTemplatePageFromTypeCS : ContentPage
{
    public WithDataTemplatePageFromTypeCS()
    {
        ...
        var people = new List<Person>
        {
            new Person { Name = "Steve", Age = 21, Location = "USA" },
            ...
        };

        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children = {
                ...
                new ListView { ItemTemplate = new DataTemplate(typeof(PersonCellCS)), ItemsSource = people,
Margin = new Thickness(0, 20, 0, 0) }
            }
        };
    }
}

```

In C#, the `DataTemplate` is created using a constructor overload that specifies the cell type as an argument. The cell type must derive from type `ViewCell`, as shown in the following code example:

```

public class PersonCellCS : ViewCell
{
    public PersonCellCS()
    {
        var grid = new Grid();
        ...
        var nameLabel = new Label { FontAttributes = FontAttributes.Bold };
        var ageLabel = new Label();
        var locationLabel = new Label { HorizontalTextAlignment = TextAlignment.End };

        nameLabel.SetBinding(Label.TextProperty, "Name");
        ageLabel.SetBinding(Label.TextProperty, "Age");
        locationLabel.SetBinding(Label.TextProperty, "Location");

        grid.Children.Add(nameLabel);
        grid.Children.Add(ageLabel, 1, 0);
        grid.Children.Add(locationLabel, 2, 0);

        View = grid;
    }
}

```

#### NOTE

Note that Xamarin.Forms also includes cell types that can be used to display simple data in `ListView` cells. For more information, see [Cell Appearance](#).

## Creating a DataTemplate as a Resource

Data templates can also be created as reusable objects in a `ResourceDictionary`. This is achieved by giving each declaration a unique `x:Key` attribute, which provides it with a descriptive key in the `ResourceDictionary`, as shown in the following XAML code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    ...
    </ContentPage.Resources>
    <ResourceDictionary>
        <DataTemplate x:Key="personTemplate">
            <ViewCell>
                <Grid>
                    ...
                </Grid>
            </ViewCell>
        </DataTemplate>
    </ResourceDictionary>
</ContentPage.Resources>
<StackLayout Margin="20">
    ...
    <ListView ItemTemplate="{StaticResource personTemplate}" Margin="0,20,0,0">
        <ListView.ItemsSource>
            <x:Array Type="{x:Type local:Person}">
                <local:Person Name="Steve" Age="21" Location="USA" />
                ...
            </x:Array>
        </ListView.ItemsSource>
    </ListView>
</StackLayout>
</ContentPage>

```

The `DataTemplate` is assigned to the `ListView.ItemTemplate` property by using the `StaticResource` markup extension. Note that while the `DataTemplate` is defined in the page's `ResourceDictionary`, it could also be defined at the control level or application level.

The following code example shows the equivalent page in C#:

```

public class WithDataTemplatePageCS : ContentPage
{
    public WithDataTemplatePageCS ()
    {
        ...
        var personDataTemplate = new DataTemplate (() => {
            var grid = new Grid ();
            ...
            return new ViewCell { View = grid };
        });

        Resources = new ResourceDictionary ();
        Resources.Add ("personTemplate", personDataTemplate);

        Content = new StackLayout {
            Margin = new Thickness(20),
            Children = {
                ...
                new ListView { ItemTemplate = (DataTemplate)Resources ["personTemplate"], ItemsSource = people };
            }
        };
    }
}

```

The `DataTemplate` is added to the `ResourceDictionary` using the `Add` method, which specifies a `key` string that is used to reference the `DataTemplate` when retrieving it.

## Summary

This article has explained how to create data templates, inline, from a custom type, or in a [ResourceDictionary](#). An inline template should be used if there's no need to reuse the data template elsewhere. Alternatively, a data template can be reused by defining it as a custom type, or as a control-level, page-level, or application-level resource.

## Related Links

- [Cell Appearance](#)
- [Data Templates \(sample\)](#)
- [DataTemplate](#)

# Creating a Xamarin.Forms DataTemplateSelector

7/12/2018 • 3 minutes to read • [Edit Online](#)

A `DataTemplateSelector` can be used to choose a `DataTemplate` at runtime based on the value of a data-bound property. This enables multiple `DataTemplates` to be applied to the same type of object, to customize the appearance of particular objects. This article demonstrates how to create and consume a `DataTemplateSelector`.

A data template selector enables scenarios such as a `ListView` binding to a collection of objects, where the appearance of each object in the `ListView` can be chosen at runtime by the data template selector returning a particular `DataTemplate`.

## Creating a DataTemplateSelector

A data template selector is implemented by creating a class that inherits from `DataTemplateSelector`. The `OnSelectTemplate` method is then overridden to return a particular `DataTemplate`, as shown in the following code example:

```
public class PersonDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate ValidTemplate { get; set; }
    public DataTemplate InvalidTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate (object item, BindableObject container)
    {
        return ((Person)item).DateOfBirth.Year >= 1980 ? ValidTemplate : InvalidTemplate;
    }
}
```

The `OnSelectTemplate` method returns the appropriate template based on the value of the `DateOfBirth` property. The template to return is the value of the `ValidTemplate` property or the `InvalidTemplate` property, which are set when consuming the `PersonDataTemplateSelector`.

An instance of the data template selector class can then be assigned to Xamarin.Forms control properties such as `ListView.ItemTemplate`. For a list of valid properties, see [Creating a DataTemplate](#).

## Limitations

`DataTemplateSelector` instances have the following limitations:

- The `DataTemplateSelector` subclass must always return the same template for the same data if queried multiple times.
- The `DataTemplateSelector` subclass must not return another `DataTemplateSelector` subclass.
- The `DataTemplateSelector` subclass must not return new instances of a `DataTemplate` on each call. Instead, the same instance must be returned. Failure to do so will create a memory leak and will disable virtualization.
- On Android, there can be no more than 20 different data templates per `ListView`.

## Consuming a DataTemplateSelector in XAML

In XAML, the `PersonDataTemplateSelector` can be instantiated by declaring it as a resource, as shown in the following code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Selector;assembly=Selector"
    x:Class="Selector.HomePage">
    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="validPersonTemplate">
                <ViewCell>
                    ...
                </ViewCell>
            </DataTemplate>
            <DataTemplate x:Key="invalidPersonTemplate">
                <ViewCell>
                    ...
                </ViewCell>
            </DataTemplate>
            <local:PersonDataTemplateSelector x:Key="personDataTemplateSelector"
                ValidTemplate="{StaticResource validPersonTemplate}"
                InvalidTemplate="{StaticResource invalidPersonTemplate}" />
        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>

```

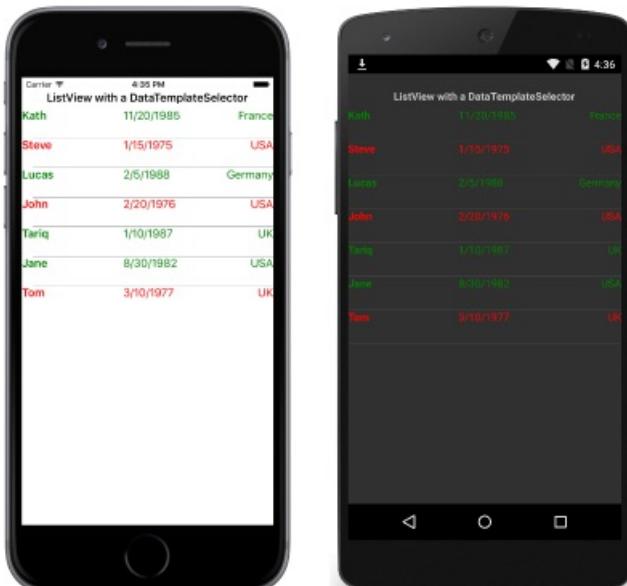
This page level `ResourceDictionary` defines two `DataTemplate` instances and a `PersonDataTemplateSelector` instance. The `PersonDataTemplateSelector` instance sets its `ValidTemplate` and `InvalidTemplate` properties to the appropriate `DataTemplate` instances by using the `StaticResource` markup extension. Note that while the resources are defined in the page's `ResourceDictionary`, they could also be defined at the control level or application level.

The `PersonDataTemplateSelector` instance is consumed by assigning it to the `ListView.ItemTemplate` property, as shown in the following code example:

```
<ListView x:Name="listView" ItemTemplate="{StaticResource personDataTemplateSelector}" />
```

At runtime, the `ListView` calls the `PersonDataTemplateSelector.OnSelectTemplate` method for each of the items in the underlying collection, with the call passing the data object as the `item` parameter. The `DataTemplate` that is returned by the method is then applied to that object.

The following screenshots show the result of the `ListView` applying the `PersonDataTemplateSelector` to each object in the underlying collection:



Any `Person` object that has a `DateOfBirth` property value greater than or equal to 1980 is displayed in green, with the remaining objects being displayed in red.

## Consuming a DataTemplateSelector in C#

In C#, the `PersonDataTemplateSelector` can be instantiated and assigned to the `ListView.ItemTemplate` property, as shown in the following code example:

```
public class HomePageCS : ContentPage
{
    DataTemplate validTemplate;
    DataTemplate invalidTemplate;

    public HomePageCS ()
    {
        ...
        SetupDataTemplates ();
        var listView = new ListView {
            ItemsSource = people,
            ItemTemplate = new PersonDataTemplateSelector {
                ValidTemplate = validTemplate,
                InvalidTemplate = invalidTemplate }
        };
        Content = new StackLayout {
            Margin = new Thickness (20),
            Children = {
                ...
                listView
            }
        };
    }
    ...
}
```

The `PersonDataTemplateSelector` instance sets its `ValidTemplate` and `InvalidTemplate` properties to the appropriate `DataTemplate` instances created by the `SetupDataTemplates` method. At runtime, the `ListView` calls the `PersonDataTemplateSelector.OnSelectTemplate` method for each of the items in the underlying collection, with the call passing the data object as the `item` parameter. The `DataTemplate` that is returned by the method is then applied to that object.

## Summary

This article has demonstrated how to create and consume a `DataTemplateSelector`. A `DataTemplateSelector` can be used to choose a `DataTemplate` at runtime based on the value of a data-bound property. This enables multiple `DataTemplate` instances to be applied to the same type of object, to customize the appearance of particular objects.

## Related Links

- [Data Template Selector \(sample\)](#)
- [DataTemplateSelector](#)

# Xamarin.Forms Triggers

10/31/2018 • 6 minutes to read • [Edit Online](#)

Triggers allow you to express actions declaratively in XAML that change the appearance of controls based on events or property changes.

You can assign a trigger directly to a control, or add it to a page-level or app-level resource dictionary to be applied to multiple controls.

There are four types of trigger:

- [Property Trigger](#) - occurs when a property on a control is set to a particular value.
- [Data Trigger](#) - uses data binding to trigger based on the properties of another control.
- [Event Trigger](#) - occurs when an event occurs on the control.
- [Multi Trigger](#) - allows multiple trigger conditions to be set before an action occurs.

## Property Triggers

A simple trigger can be expressed purely in XAML, adding a `Trigger` element to a control's triggers collection. This example shows a trigger that changes an `Entry` background color when it receives focus:

```
<Entry Placeholder="enter name">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
            Property="IsFocused" Value="True">
            <Setter Property="BackgroundColor" Value="Yellow" />
        </Trigger>
    </Entry.Triggers>
</Entry>
```

The important parts of the trigger's declaration are:

- **TargetType** - the control type that the trigger applies to.
- **Property** - the property on the control that is monitored.
- **Value** - the value, when it occurs for the monitored property, that causes the trigger to activate.
- **Setter** - a collection of `Setter` elements can be added and when the trigger condition is met. You must specify the `Property` and `Value` to set.
- **EnterActions and ExitActions** (not shown) - are written in code and can be used in addition to (or instead of) `Setter` elements. They are [described below](#).

## Applying a Trigger using a Style

Triggers can also be added to a `Style` declaration on a control, in a page, or an application `ResourceDictionary`. This example declares an implicit style (ie. no `key` is set) which means it will apply to all `Entry` controls on the page.

```

<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Entry">
            <Style.Triggers>
                <Trigger TargetType="Entry"
                    Property="IsFocused" Value="True">
                    <Setter Property="BackgroundColor" Value="Yellow" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

```

## Data Triggers

Data triggers use data binding to monitor another control to cause the `Setter`s to get called. Instead of the `Property` attribute in a property trigger, set the `Binding` attribute to monitor for the specified value.

The example below uses the data binding syntax `{Binding Source={x:Reference entry}, Path=Text.Length}` which is how we refer to another control's properties. When the length of the `entry` is zero, the trigger is activated. In this sample the trigger disables the button when the input is empty.

```

<!-- the x:Name is referenced below in DataTrigger-->
<!-- tip: make sure to set the Text="" (or some other default) -->
<Entry x:Name="entry"
    Text=""
    Placeholder="required field" />

<Button x:Name="button" Text="Save"
    FontSize="Large"
    HorizontalOptions="Center">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
            Binding="{Binding Source={x:Reference entry},
                Path=Text.Length}"
            Value="0">
            <Setter Property="IsEnabled" Value="False" />
        </DataTrigger>
    </Button.Triggers>
</Button>

```

Tip: when evaluating `Path=Text.Length` always provide a default value for the target property (eg. `Text=""`) because otherwise it will be `null` and the trigger won't work like you expect.

In addition to specifying `Setter`s you can also provide `EnterActions` and `ExitActions`.

## Event Triggers

The `EventTrigger` element requires only an `Event` property, such as `"Clicked"` in the example below.

```

<EventTrigger Event="Clicked">
    <local:NumericValidationTriggerAction />
</EventTrigger>

```

Notice that there are no `Setter` elements but rather a reference to a class defined by `local:NumericValidationTriggerAction` which requires the `xmlns:local` to be declared in the page's XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:WorkingWithTriggers;assembly=WorkingWithTriggers"
```

The class itself implements `TriggerAction` which means it should provide an override for the `Invoke` method that is called whenever the trigger event occurs.

A trigger action implementation should:

- Implement the generic `TriggerAction<T>` class, with the generic parameter corresponding with the type of control the trigger will be applied to. You can use superclasses such as `VisualElement` to write trigger actions that work with a variety of controls, or specify a control type like `Entry`.
- Override the `Invoke` method - this is called whenever the trigger criteria are met.
- Optionally expose properties that can be set in the XAML when the trigger is declared (such as `Anchor`, `Scale`, and `Length` in this example).

```
public class NumericValidationTriggerAction : TriggerAction<Entry>
{
    protected override void Invoke (Entry entry)
    {
        double result;
        bool isValid = Double.TryParse (entry.Text, out result);
        entry.TextColor = isValid ? Color.Default : Color.Red;
    }
}
```

The properties exposed by the trigger action can be set in the XAML declaration as follows:

```
<EventTrigger Event="TextChanged">
    <local:NumericValidationTriggerAction />
</EventTrigger>
```

Be careful when sharing triggers in a `ResourceDictionary`, one instance will be shared among controls so any state that is configured once will apply to them all.

Note that event triggers do not support `EnterActions` and `ExitActions` [described below](#).

## Multi Triggers

A `MultiTrigger` looks similar to a `Trigger` or `DataTrigger` except there can be more than one condition. All the conditions must be true before the `Setter`'s are triggered.

Here's an example of a trigger for a button that binds to two different inputs (`email` and `phone`):

```

<MultiTrigger TargetType="Button">
    <MultiTrigger.Conditions>
        <BindingCondition Binding="{Binding Source={x:Reference email},
                                         Path=Text.Length}"
                           Value="0" />
        <BindingCondition Binding="{Binding Source={x:Reference phone},
                                         Path=Text.Length}"
                           Value="0" />
    </MultiTrigger.Conditions>

    <Setter Property="IsEnabled" Value="False" />
    <!-- multiple Setter elements are allowed -->
</MultiTrigger>

```

The `Conditions` collection could also contain `PropertyCondition` elements like this:

```
<PropertyCondition Property="Text" Value="OK" />
```

### Building a "require all" multi trigger

The multi trigger only updates its control when all conditions are true. Testing for "all field lengths are zero" (such as a login page where all inputs must be complete) is tricky because you want a condition "where `Text.Length > 0`" but this can't be expressed in XAML.

This can be done with an `IValueConverter`. The converter code below transforms the `Text.Length` binding into a `bool` that indicates whether a field is empty or not:

```

public class MultiTriggerConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                         object parameter, CultureInfo culture)
    {
        if ((int)value > 0) // length > 0 ?
            return true;           // some data has been entered
        else
            return false;          // input is empty
    }

    public object ConvertBack(object value, Type targetType,
                            object parameter, CultureInfo culture)
    {
        throw new NotSupportedException ();
    }
}

```

To use this converter in a multi trigger, first add it to the page's resource dictionary (along with a custom `xmlns:local` namespace definition):

```

<ResourceDictionary>
    <local:MultiTriggerConverter x:Key="dataHasBeenEntered" />
</ResourceDictionary>

```

The XAML is shown below. Note the following differences from the first multi trigger example:

- The button has `.IsEnabled="false"` set by default.
- The multi trigger conditions use the converter to turn the `Text.Length` value into a `boolean`.
- When all the conditions are `true`, the setter makes the button's `.IsEnabled` property `true`.

```

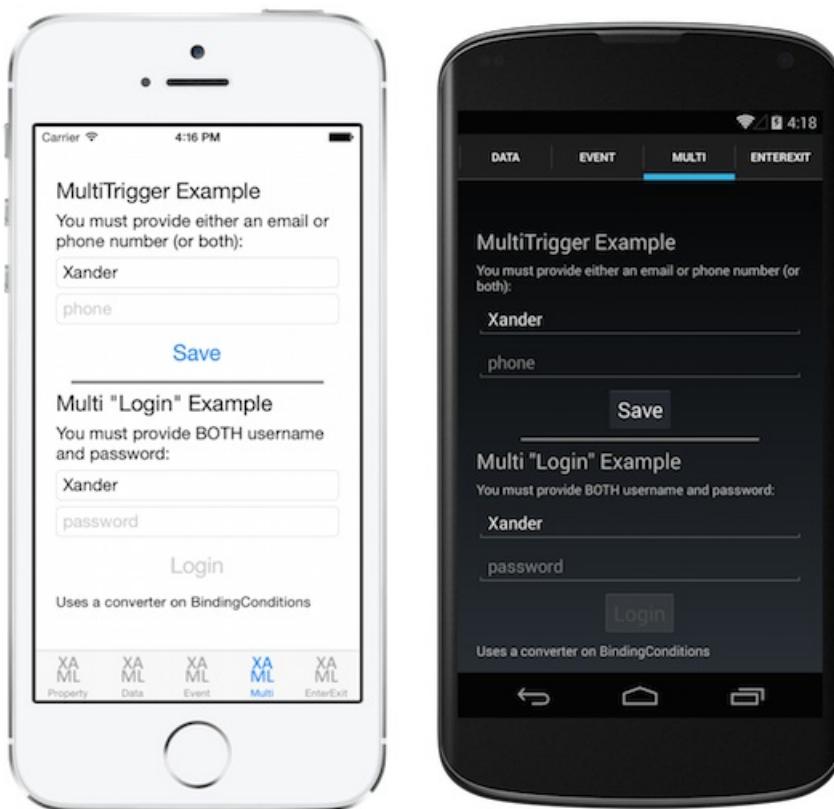
<Entry x:Name="user" Text="" Placeholder="user name" />

<Entry x:Name="pwd" Text="" Placeholder="password" />

<Button x:Name="loginButton" Text="Login"
    FontSize="Large"
    HorizontalOptions="Center"
    IsEnabled="false">
    <Button.Triggers>
        <MultiTrigger TargetType="Button">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference user},
                    Path=Text.Length,
                    Converter={StaticResource dataHasBeenEntered}}"
                    Value="true" />
                <BindingCondition Binding="{Binding Source={x:Reference pwd},
                    Path=Text.Length,
                    Converter={StaticResource dataHasBeenEntered}}"
                    Value="true" />
            </MultiTrigger.Conditions>
            <Setter Property="IsEnabled" Value="True" />
        </MultiTrigger>
    </Button.Triggers>
</Button>

```

These screenshots show the difference between the two multi trigger examples above. In the top part of the screens, text input in just one `Entry` is enough to enable the **Save** button. In the bottom part of the screens, the **Login** button remains inactive until both fields contain data.



## EnterActions and ExitActions

Another way to implement changes when a trigger occurs is by adding `EnterActions` and `ExitActions` collections and specifying `TriggerAction<T>` implementations.

You can provide *both* `EnterActions` and `ExitActions` as well as `Setter`s in a trigger, but be aware that the `Setter`s are called immediately (they do not wait for the `EnterAction` or `ExitAction` to complete). Alternatively

you can perform everything in the code and not use `Setter`s at all.

```
<Entry Placeholder="enter job title">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
            Property="Entry.IsFocused" Value="True">
            <Trigger.EnterActions>
                <local:FadeTriggerAction StartsFrom="0"" />
            </Trigger.EnterActions>

            <Trigger.ExitActions>
                <local:FadeTriggerAction StartsFrom="1" />
            </Trigger.ExitActions>
            <!-- You can use both Enter/Exit and Setter together if required -->
        </Trigger>
    </Entry.Triggers>
</Entry>
```

As always, when a class is referenced in XAML you should declare a namespace such as `xmlns:local` as shown here:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:WorkingWithTriggers;assembly=WorkingWithTriggers"
```

The `FadeTriggerAction` code is shown below:

```
public class FadeTriggerAction : TriggerAction<VisualElement>
{
    public FadeTriggerAction() {}

    public int StartsFrom { set; get; }

    protected override void Invoke (VisualElement visual)
    {
        visual.Animate("", new Animation( (d)=>{
            var val = StartsFrom==1 ? d : 1-d;
            visual.BackgroundColor = Color.FromRgb(1, val, 1);

        }),
        length:1000, // milliseconds
        easing: Easing.Linear);
    }
}
```

Note: `EnterActions` and `ExitActions` are ignored on **Event Triggers**.

## Related Links

- [Triggers Sample](#)
- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms User Interface Views

11/20/2018 • 2 minutes to read • [Edit Online](#)

*How to use the views provided by Xamarin.Forms*

## Animation

Xamarin.Forms includes its own animation infrastructure that's straightforward for creating simple animations, while also being versatile enough to create complex animations.

## BoxView

The `BoxView` is just a simple colored rectangle, but it can be used for decorative items, rudimentary graphics, and for obtaining interactive touch input.

## Button

The `Button` responds to a tap or click that directs an application to carry out a particular task.

## Colors

Defining and using colors across platforms can be tricky when each platform has its own standards and defaults.

## Controls Reference

This document is a quick reference to the UI views that make up the Xamarin.Forms framework, such as [Pages](#), [Layouts](#), [Views](#) and [Cells](#).

## DataPages

DataPages provide an API to quickly and easily bind a data source to pre-built views. List items and detail pages will automatically render the data, and be customized using themes.

## DatePicker

The `DatePicker` allows a user to select a date within a specified range. It is implemented using the date picker supported by the particular platform that the application is run on.

## Graphics with SkiaSharp

How to incorporate graphics into a Xamarin.Forms application using SkiaSharp.

## Images

Images can be shared across platforms with Xamarin.Forms, they can be loaded specifically for each platform, or they can be downloaded for display.

## ImageButton

The `ImageButton` displays an image and responds to a tap or click that directs an application to carry out a

particular task.

## Layouts

Xamarin.Forms has several layouts for organizing on-screen content. `StackLayout`, `Grid`, `FlexLayout`, `AbsoluteLayout`, `ScrollView`, and `RelativeLayout` can each be used to create beautiful, responsive user interfaces.

## ListView

Xamarin.Forms provides a list view control to display scrolling rows of data. The control includes contextual actions, `HasUnevenRows` automatic sizing, separator customization, pull-to-refresh, and headers and footers.

## Maps

Adding maps requires an additional NuGet package download and some platform-specific configuration. Maps and pin markers can be added in just a few lines of code once the configuration is done.

## Picker

The `Picker` view is a control for selecting a text item from a list of data.

## Slider

The `Slider` allows a user to select a numeric value from a continuous range.

## Stepper

The `Stepper` allows a user to select a numeric value from a range of values. It consists of two buttons labeled with minus and plus signs. Manipulating the two buttons changes the selected value incrementally.

## Styles

Font, color, and other attributes can be grouped into styles which can be shared across controls, layouts, or the entire application using ResourceDictionaries.

## TableView

The table view is similar to a list view, but rather than being designed for long lists of data it is intended for data-entry-style screens of scrolling controls or simple scrolling menus.

## Text

Xamarin.Forms has several views for presenting and receiving text. Text views can be formatted and customized for platforms. Specific font settings can enable compatibility with accessibility features.

## Themes

Xamarin.Forms Themes define a specific visual appearance for the standard controls. Once you add a theme to the application's resource dictionary, the appearance of the standard controls will change.

## TimePicker

The `TimePicker` allows a user to select a time. It is implemented using the time picker supported by the particular platform that the application is run on.

## Visual State Manager

The Visual State Manager provides a structured way to trigger changes in the user interface from code, including layout that adapts to changes in device orientation or size.

## WebView

Xamarin.Forms uses the native web browser control on each platform, and can display websites, local resources, and generated Html strings.

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms Gallery \(sample\)](#)

# Animation in Xamarin.Forms

7/12/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms includes its own animation infrastructure that's straightforward for creating simple animations, while also being versatile enough to create complex animations.

The Xamarin.Forms animation classes target different properties of visual elements, with a typical animation progressively changing a property from one value to another over a period of time. Note that there is no XAML interface for the Xamarin.Forms animation classes. However, animations can be encapsulated in [behaviors](#) and then referenced from XAML.

## Simple Animations

The [ViewExtensions](#) class provides extension methods that can be used to construct simple animations that rotate, scale, translate, and fade [VisualElement](#) instances. This article demonstrates creating and canceling animations using the [ViewExtensions](#) class.

## Easing Functions

Xamarin.Forms includes an [Easing](#) class that allows you to specify a transfer function that controls how animations speed up or slow down as they're running. This article demonstrates how to consume the pre-defined easing functions, and how to create custom easing functions.

## Custom Animations

The [Animation](#) class is the building block of all Xamarin.Forms animations, with the extension methods in the [ViewExtensions](#) class creating one or more [Animation](#) objects. This article demonstrates how to use the [Animation](#) class to create and cancel animations, synchronize multiple animations, and create custom animations that animate properties that aren't animated by the existing animation methods.

# Simple Animations in Xamarin.Forms

9/20/2018 • 9 minutes to read • [Edit Online](#)

The `ViewExtensions` class provides extension methods that can be used to construct simple animations. This article demonstrates creating and canceling animations using the `ViewExtensions` class.

The `ViewExtensions` class provides the following extension methods that can be used to create simple animations:

- `TranslateTo` animates the `TranslationX` and `TranslationY` properties of a `VisualElement`.
- `ScaleTo` animates the `Scale` property of a `VisualElement`.
- `RelScaleTo` applies an animated incremental increase or decrease to the `Scale` property of a `VisualElement`.
- `RotateTo` animates the `Rotation` property of a `VisualElement`.
- `RelRotateTo` applies an animated incremental increase or decrease to the `Rotation` property of a `VisualElement`.
- `RotateXTo` animates the `RotationX` property of a `VisualElement`.
- `RotateYTo` animates the `RotationY` property of a `VisualElement`.
- `FadeTo` animates the `Opacity` property of a `VisualElement`.

By default, each animation will take 250 milliseconds. However, a duration for each animation can be specified when creating the animation.

The `ViewExtensions` class also includes a `CancelAnimations` method that can be used to cancel any animations.

## NOTE

The `ViewExtensions` class provides a `LayoutTo` extension method. However, this method is intended to be used by layouts to animate transitions between layout states that contain size and position changes. Therefore, it should only be used by `Layout` subclasses.

The animation extension methods in the `ViewExtensions` class are all asynchronous and return a `Task<bool>` object. The return value is `false` if the animation completes, and `true` if the animation is cancelled. Therefore, the animation methods should typically be used with the `await` operator, which makes it possible to easily determine when an animation has completed. In addition, it then becomes possible to create sequential animations with subsequent animation methods executing after the previous method has completed. For more information, see [Compound Animations](#).

If there's a requirement to let an animation complete in the background, then the `await` operator can be omitted. In this scenario, the animation extension methods will quickly return after initiating the animation, with the animation occurring in the background. This operation can be taken advantage of when creating composite animations. For more information, see [Composite Animations](#).

For more information about the `await` operator, see [Async Support Overview](#).

## Single Animations

Each extension method in the `ViewExtensions` implements a single animation operation that progressively changes a property from one value to another value over a period of time. This section explores each animation operation.

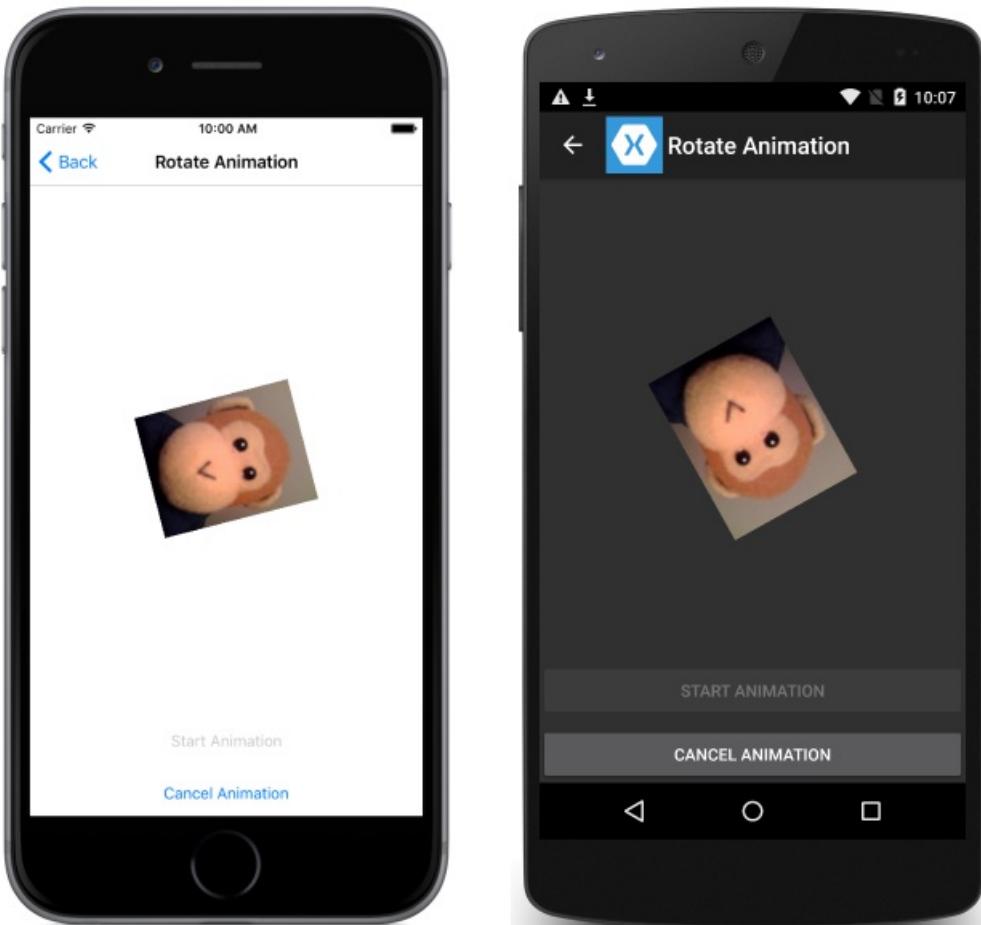
### Rotation

The following code example demonstrates using the `RotateTo` method to animate the `Rotation` property of an `Image`:

```
await image.RotateTo (360, 2000);
image.Rotation = 0;
```

This code animates the `Image` instance by rotating up to 360 degrees over 2 seconds (2000 milliseconds). The `RotateTo` method obtains the current `Rotation` property value for the start of the animation, and then rotates from that value to its first argument (360). Once the animation is complete, the image's `Rotation` property is reset to 0. This ensures that the `Rotation` property doesn't remain at 360 after the animation concludes, which would prevent additional rotations.

The following screenshots show the rotation in progress on each platform:



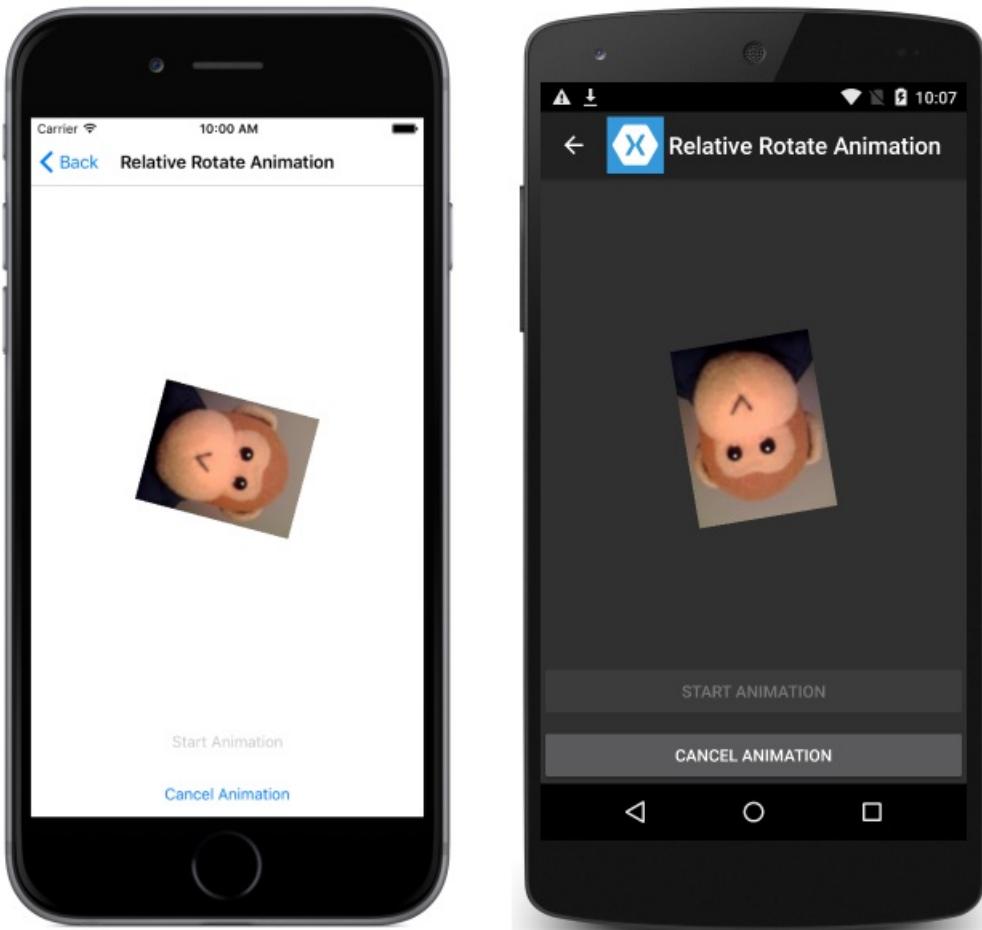
## Relative Rotation

The following code example demonstrates using the `RelRotateTo` method to incrementally increase or decrease the `Rotation` property of an `Image`:

```
await image.RelRotateTo (360, 2000);
```

This code animates the `Image` instance by rotating 360 degrees from its starting position over 2 seconds (2000 milliseconds). The `RelRotateTo` method obtains the current `Rotation` property value for the start of the animation, and then rotates from that value to the value plus its first argument (360). This ensures that each animation will always be a 360 degrees rotation from the starting position. Therefore, if a new animation is invoked while an animation is already in progress, it will start from the current position and may end at a position that is not an increment of 360 degrees.

The following screenshots show the relative rotation in progress on each platform:



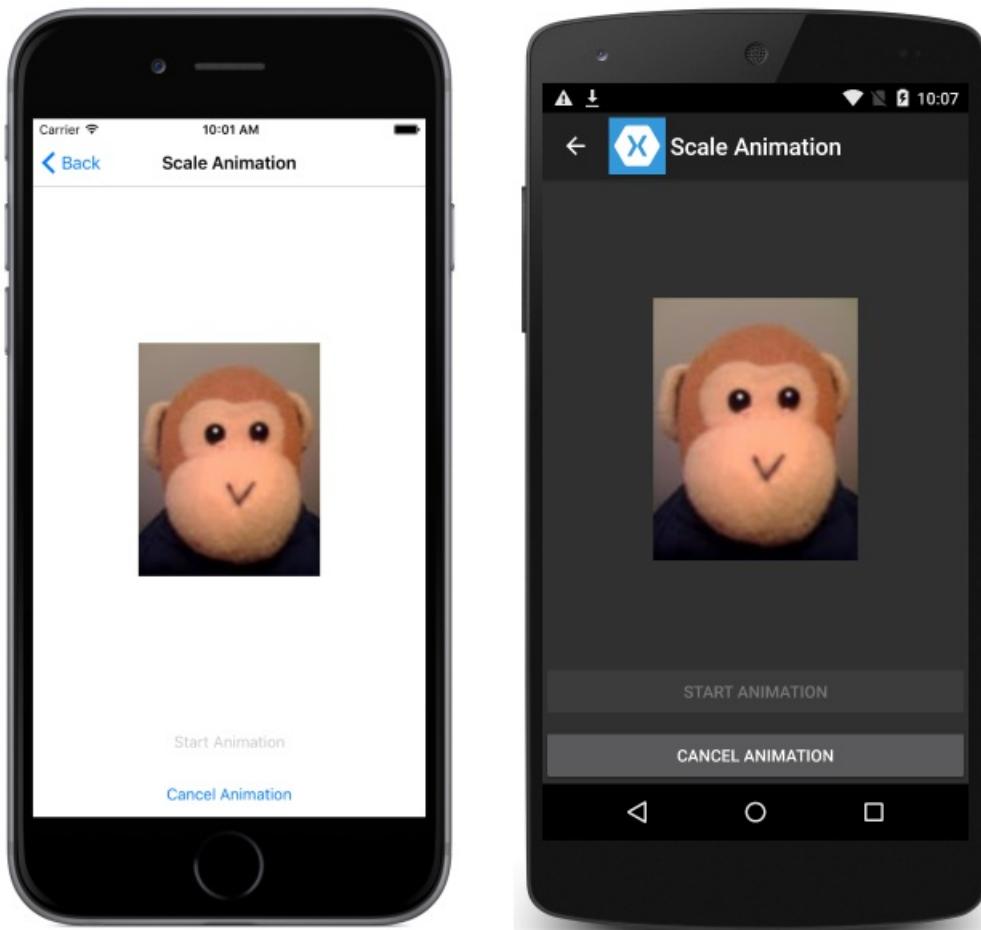
## Scaling

The following code example demonstrates using the `ScaleTo` method to animate the `Scale` property of an `Image`:

```
await image.ScaleTo (2, 2000);
```

This code animates the `Image` instance by scaling up to twice its size over 2 seconds (2000 milliseconds). The `ScaleTo` method obtains the current `Scale` property value (default value of 1) for the start of the animation, and then scales from that value to its first argument (2). This has the effect of expanding the size of the image to twice its size.

The following screenshots show the scaling in progress on each platform:



#### NOTE

The `VisualElement` class also defines `ScaleX` and `ScaleY` properties, which can scale the `VisualElement` differently in the horizontal and vertical directions. These properties can be animated with the `Animation` class. For more information, see [Custom Animations in Xamarin.Forms](#).

### Relative Scaling

The following code example demonstrates using the `RelScaleTo` method to animate the `Scale` property of an `Image`:

```
await image.RelScaleTo (2, 2000);
```

This code animates the `Image` instance by scaling up to twice its size over 2 seconds (2000 milliseconds). The `RelScaleTo` method obtains the current `Scale` property value for the start of the animation, and then scales from that value to the value plus its first argument (2). This ensures that each animation will always be a scaling of 2 from the starting position.

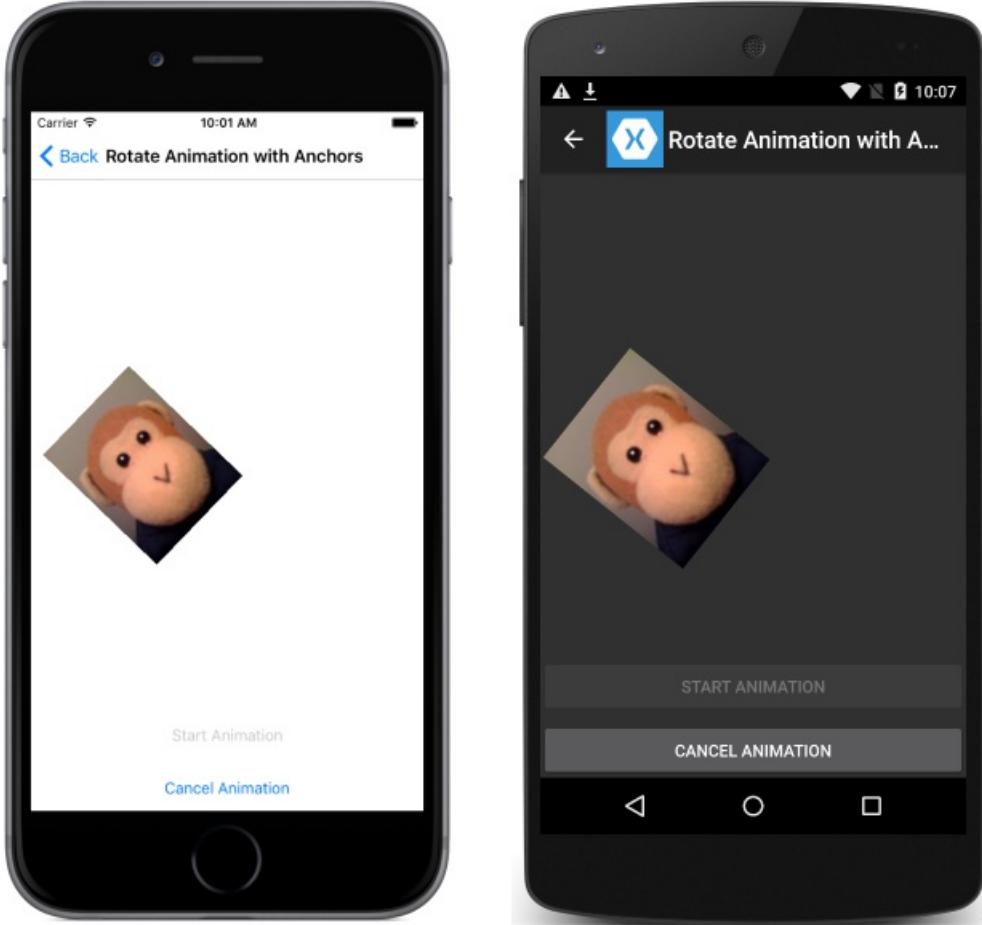
### Scaling and Rotation with Anchors

The `AnchorX` and `AnchorY` properties set the center of scaling or rotation for the `Rotation` and `Scale` properties. Therefore, their values also affect the `RotateTo` and `ScaleTo` methods.

Given an `Image` that has been placed at the center of a layout, the following code example demonstrates rotating the image around the center of the layout by setting its `AnchorY` property:

```
image.AnchorY = (Math.Min (absoluteLayout.Width, absoluteLayout.Height) / 2) / image.Height;  
await image.RotateTo (360, 2000);
```

To rotate the `Image` instance around the center of the layout, the `AnchorX` and `AnchorY` properties must be set to values that are relative to the width and height of the `Image`. In this example, the center of the `Image` is defined to be at the center of the layout, and so the default `AnchorX` value of 0.5 does not require changing. However, the `AnchorY` property is redefined to be a value from the top of the `Image` to the center point of the layout. This ensures that the `Image` makes a full rotation of 360 degrees around the center point of the layout, as shown in the following screenshots:



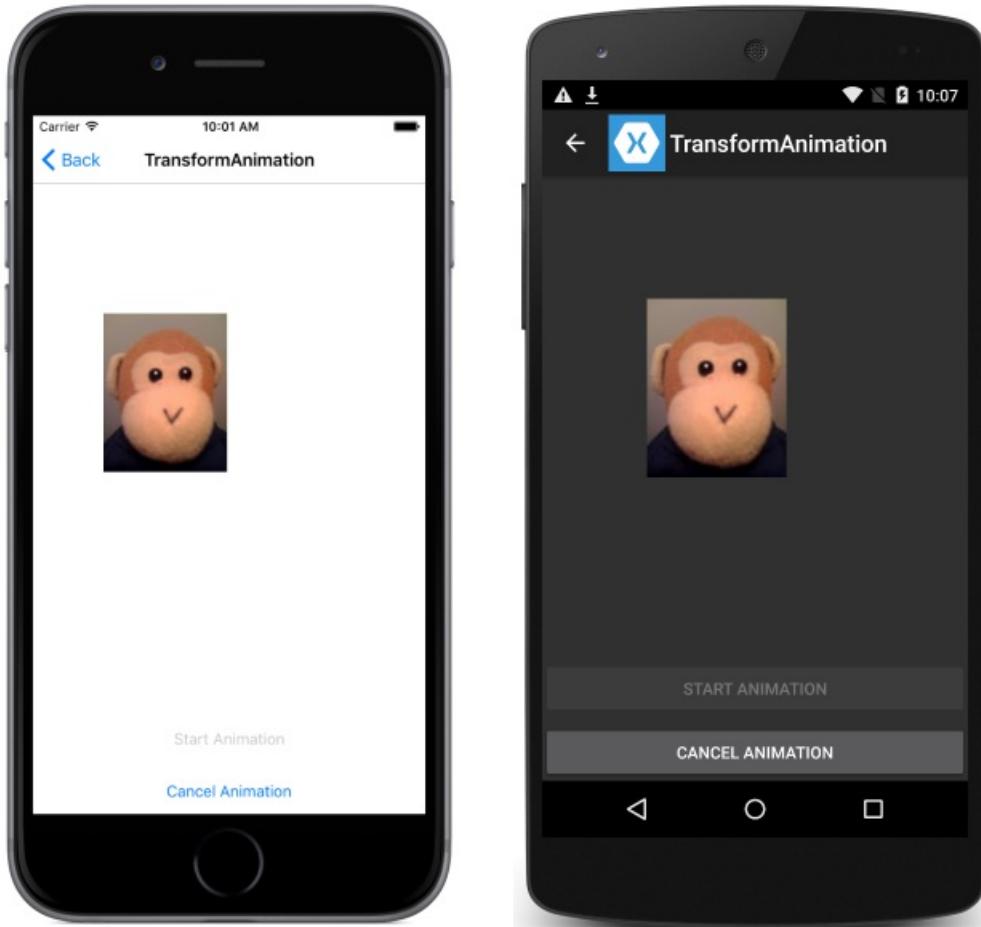
## Translation

The following code example demonstrates using the `TranslateTo` method to animate the `TranslationX` and `TranslationY` properties of an `Image`:

```
await image.TranslateTo (-100, -100, 1000);
```

This code animates the `Image` instance by translating it horizontally and vertically over 1 second (1000 milliseconds). The `TranslateTo` method simultaneously translates the image 100 pixels to the left, and 100 pixels upwards. This is because the first and second arguments are both negative numbers. Providing positive numbers would translate the image to the right, and down.

The following screenshots show the translation in progress on each platform:



#### NOTE

If an element is initially laid out off screen and then translated onto the screen, after translation the element's input layout remains off screen and the user can't interact with it. Therefore, it's recommended that a view should be laid out in its final position, and then any required translations performed.

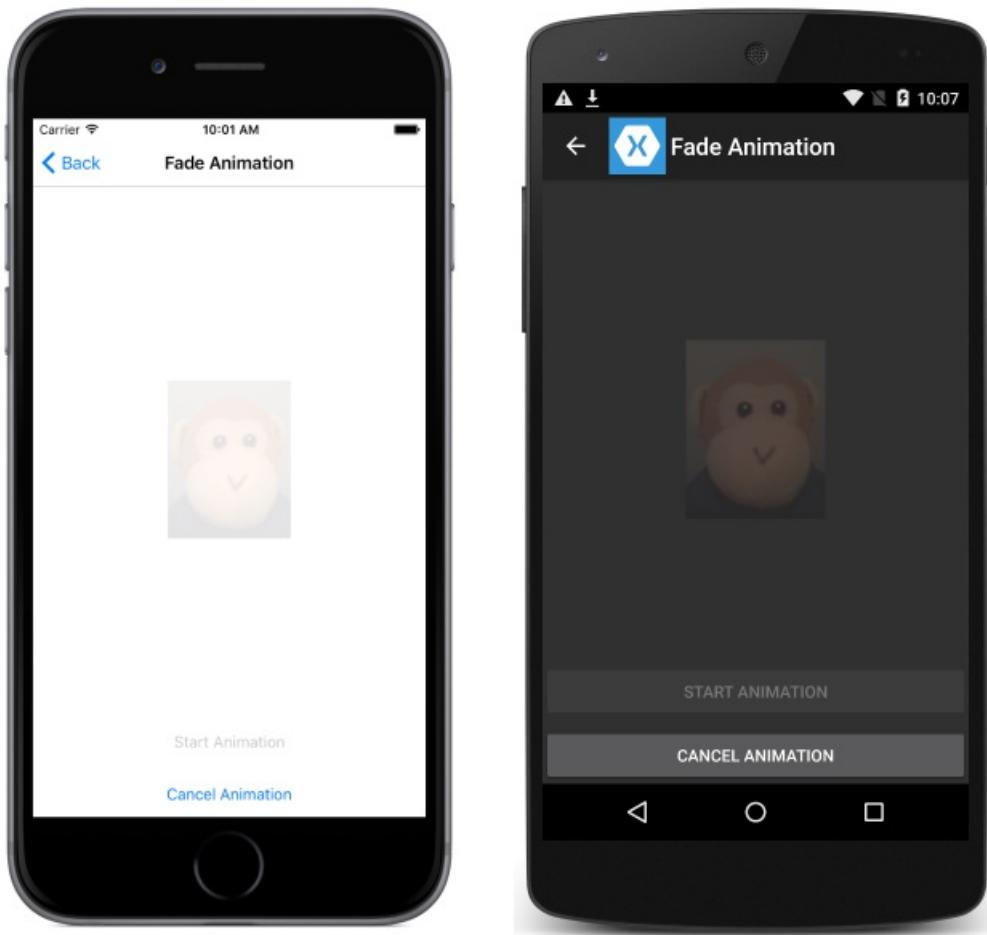
#### Fading

The following code example demonstrates using the `FadeTo` method to animate the `Opacity` property of an `Image`:

```
image.Opacity = 0;  
await image.FadeTo (1, 4000);
```

This code animates the `Image` instance by fading it in over 4 seconds (4000 milliseconds). The `FadeTo` method obtains the current `Opacity` property value for the start of the animation, and then fades in from that value to its first argument (1).

The following screenshots show the fade in progress on each platform:



## Compound Animations

A compound animation is a sequential combination of animations, and can be created with the `await` operator, as demonstrated in the following code example:

```
await image.TranslateTo (-100, 0, 1000);    // Move image left
await image.TranslateTo (-100, -100, 1000); // Move image up
await image.TranslateTo (100, 100, 2000);   // Move image diagonally down and right
await image.TranslateTo (0, 100, 1000);      // Move image left
await image.TranslateTo (0, 0, 1000);        // Move image up
```

In this example, the `Image` is translated over 6 seconds (6000 milliseconds). The translation of the `Image` uses five animations, with the `await` operator indicating that each animation executes sequentially. Therefore, subsequent animation methods execute after the previous method has completed.

## Composite Animations

A composite animation is a combination of animations where two or more animations run simultaneously. Composite animations can be created by mixing awaited and non-awaited animations, as demonstrated in the following code example:

```
image.RotateTo (360, 4000);
await image.ScaleTo (2, 2000);
await image.ScaleTo (1, 2000);
```

In this example, the `Image` is scaled and simultaneously rotated over 4 seconds (4000 milliseconds). The scaling of the `Image` uses two sequential animations that occur at the same time as the rotation. The `RotateTo` method

executes without an `await` operator and returns immediately, with the first `ScaleTo` animation then beginning. The `await` operator on the first `ScaleTo` method call delays the second `ScaleTo` method call until the first `ScaleTo` method call has completed. At this point the `RotateTo` animation is half way completed and the `Image` will be rotated 180 degrees. During the final 2 seconds (2000 milliseconds), the second `ScaleTo` animation and the `RotateTo` animation both complete.

## Running Multiple Asynchronous Methods Concurrently

The `static Task.WhenAny` and `Task.WhenAll` methods are used to run multiple asynchronous methods concurrently, and therefore can be used to create composite animations. Both methods return a `Task` object and accept a collection of methods that each return a `Task` object. The `Task.WhenAny` method completes when any method in its collection completes execution, as demonstrated in the following code example:

```
await Task.WhenAny<bool>
(
    image.RotateTo (360, 4000),
    image.ScaleTo (2, 2000)
);
await image.ScaleTo (1, 2000);
```

In this example, the `Task.WhenAny` method call contains two tasks. The first task rotates the image over 4 seconds (4000 milliseconds), and the second task scales the image over 2 seconds (2000 milliseconds). When the second task completes, the `Task.WhenAny` method call completes. However, even though the `RotateTo` method is still running, the second `ScaleTo` method can begin.

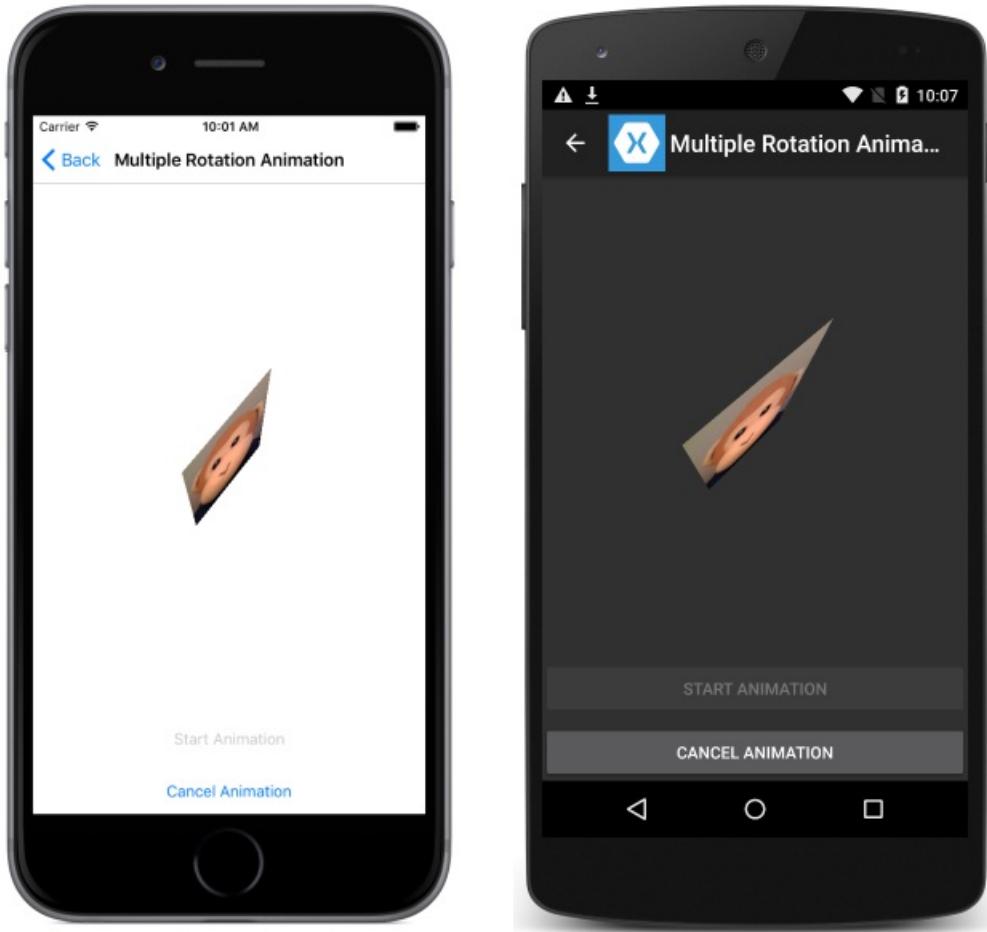
The `Task.WhenAll` method completes when all the methods in its collection have completed, as demonstrated in the following code example:

```
// 10 minute animation
uint duration = 10 * 60 * 1000;

await Task.WhenAll (
    image.RotateTo (307 * 360, duration),
    image.RotateXTo (251 * 360, duration),
    image.RotateYTo (199 * 360, duration)
);
```

In this example, the `Task.WhenAll` method call contains three tasks, each of which executes over 10 minutes. Each `Task` makes a different number of 360 degree rotations – 307 rotations for `RotateTo`, 251 rotations for `RotateXTo`, and 199 rotations for `RotateYTo`. These values are prime numbers, therefore ensuring that the rotations aren't synchronized and hence won't result in repetitive patterns.

The following screenshots show the multiple rotations in progress on each platform:



## Canceling Animations

An application can cancel one or more animations with a call to the `static ViewExtensions.CancelAnimations` method, as demonstrated in the following code example:

```
ViewExtensions.CancelAnimations (image);
```

This will immediately cancel all animations that are currently running on the `Image` instance.

## Summary

This article demonstrated creating and canceling animations using the `ViewExtensions` class. This class provides extension methods that can be used to construct simple animations that rotate, scale, translate, and fade `VisualElement` instances.

## Related Links

- [Async Support Overview](#)
- [Basic Animation \(sample\)](#)
- [ViewExtensions](#)

# Easing Functions in Xamarin.Forms

7/12/2018 • 3 minutes to read • [Edit Online](#)

Xamarin.Forms includes an `Easing` class that allows you to specify a transfer function that controls how animations speed up or slow down as they're running. This article demonstrates how to consume the pre-defined easing functions, and how to create custom easing functions.

The `Easing` class defines a number of easing functions that can be consumed by animations:

- The `BounceIn` easing function bounces the animation at the beginning.
- The `BounceOut` easing function bounces the animation at the end.
- The `CubicIn` easing function slowly accelerates the animation.
- The `CubicInOut` easing function accelerates the animation at the beginning, and decelerates the animation at the end.
- The `CubicOut` easing function quickly decelerates the animation.
- The `Linear` easing function uses a constant velocity, and is the default easing function.
- The `SinIn` easing function smoothly accelerates the animation.
- The `SinInOut` easing function smoothly accelerates the animation at the beginning, and smoothly decelerates the animation at the end.
- The `SinOut` easing function smoothly decelerates the animation.
- The `SpringIn` easing function causes the animation to very quickly accelerate towards the end.
- The `SpringOut` easing function causes the animation to quickly decelerate towards the end.

The `In` and `out` suffixes indicate if the effect provided by the easing function is noticeable at the beginning of the animation, at the end, or both.

In addition, custom easing functions can be created. For more information, see [Custom Easing Functions](#).

## Consuming an Easing Function

The animation extension methods in the `ViewExtensions` class allow an easing function to be specified as the final method parameter, as demonstrated in the following code example:

```
await image.TranslateTo(0, 200, 2000, Easing.BounceIn);
await image.ScaleTo(2, 2000, Easing.CubicIn);
await image.RotateTo(360, 2000, Easing.SinInOut);
await image.ScaleTo(1, 2000, Easing.CubicOut);
await image.TranslateTo(0, -200, 2000, Easing.BounceOut);
```

By specifying an easing function for an animation, the animation velocity becomes non-linear and produces the effect provided by the easing function. Omitting an easing function when creating an animation causes the animation to use the default `Linear` easing function, which produces a linear velocity.

For more information about using the animation extension methods in the `ViewExtensions` class, see [Simple Animations](#). Easing functions can also be consumed by the `Animation` class. For more information, see [Custom Animations](#).

## Custom Easing Functions

There are three main approaches to creating a custom easing function:

1. Create a method that takes a `double` argument, and returns a `double` result.
2. Create a `Func<double, double>`.
3. Specify the easing function as the argument to the `Easing` constructor.

In all three cases, the custom easing function should return 0 for an argument of 0, and 1 for an argument of 1. However, any value can be returned between the argument values of 0 and 1. Each approach will now be discussed in turn.

### Custom Easing Method

A custom easing function can be defined as a method that takes a `double` argument, and returns a `double` result, as demonstrated in the following code example:

```
await image.TranslateTo(0, 200, 2000, CustomEase);

double CustomEase (double t)
{
    return t == 0 || t == 1 ? t : (int)(5 * t) / 5.0;
}
```

The `CustomEase` method truncates the incoming value to the values 0, 0.2, 0.4, 0.6, 0.8, and 1. Therefore, the `Image` instance is translated in discrete jumps, rather than smoothly.

### Custom Easing Func

A custom easing function can also be defined as a `Func<double, double>`, as demonstrated in the following code example:

```
Func<double, double> CustomEase = t => 9 * t * t * t - 13.5 * t * t + 5.5 * t;
await image.TranslateTo(0, 200, 2000, CustomEase));
```

The `CustomEase` `Func` represents an easing function that starts off fast, slows down and reverses course, and then reverses course again to accelerate quickly towards the end. Therefore, while the overall movement of the `Image` instance is downwards, it also temporarily reverses course halfway through the animation.

### Custom Easing Constructor

A custom easing function can also be defined as the argument to the `Easing` constructor, as demonstrated in the following code example:

```
await image.TranslateTo (0, 200, 2000, new Easing (t => 1 - Math.Cos (10 * Math.PI * t) * Math.Exp (-5 * t)));
```

The custom easing function is specified as a lambda function argument to the `Easing` constructor, and uses the `Math.Cos` method to create a slow drop effect that's damped by the `Math.Exp` method. Therefore, the `Image` instance is translated so that it appears to drop to its final resting place.

## Summary

This article demonstrated how to consume the pre-defined easing functions, and how to create custom easing functions. Xamarin.Forms includes an `Easing` class that allows you to specify a transfer function that controls how animations speed up or slow down as they're running.

## Related Links

- [Async Support Overview](#)
- [Easing Functions \(sample\)](#)

- [Easing](#)
- [ViewExtensions](#)

# Custom Animations in Xamarin.Forms

7/12/2018 • 10 minutes to read • [Edit Online](#)

The `Animation` class is the building block of all `Xamarin.Forms` animations, with the extension methods in the `ViewExtensions` class creating one or more `Animation` objects. This article demonstrates how to use the `Animation` class to create and cancel animations, synchronize multiple animations, and create custom animations that animate properties that aren't animated by the existing animation methods.

A number of parameters must be specified when creating an `Animation` object, including start and end values of the property being animated, and a callback that changes the value of the property. An `Animation` object can also maintain a collection of child animations that can be run and synchronized. For more information, see [Child Animations](#).

Running an animation created with the `Animation` class, which may or may not include child animations, is achieved by calling the `Commit` method. This method specifies the duration of the animation, and amongst other items, a callback that controls whether to repeat the animation.

## Creating an Animation

When creating an `Animation` object, typically, a minimum of three parameters are required, as demonstrated in the following code example:

```
var animation = new Animation (v => image.Scale = v, 1, 2);
```

This code defines an animation of the `Scale` property of an `Image` instance from a value of 1 to a value of 2. The animated value, which is derived by `Xamarin.Forms`, is passed to the callback specified as the first argument, where it's used to change the value of the `Scale` property.

The animation is started with a call to the `Commit` method, as demonstrated in the following code example:

```
animation.Commit (this, "SimpleAnimation", 16, 2000, Easing.Linear, (v, c) => image.Scale = 1, () => true);
```

Note that the `Commit` method does not return a `Task` object. Instead, notifications are provided through callback methods.

The following arguments are specified in the `Commit` method:

- The first argument (`owner`) identifies the owner of the animation. This can be the visual element on which the animation is applied, or another visual element, such as the page.
- The second argument (`name`) identifies the animation with a name. The name is combined with the owner to uniquely identify the animation. This unique identification can then be used to determine whether the animation is running (`AnimationIsRunning`), or to cancel it (`AbortAnimation`).
- The third argument (`rate`) indicates the number of milliseconds between each call to the callback method defined in the `Animation` constructor
- The fourth argument (`length`) indicates the duration of the animation, in milliseconds.
- The fifth argument (`easing`) defines the easing function to be used in the animation. Alternatively, the easing function can be specified as an argument to the `Animation` constructor. For more information about easing functions, see [Easing Functions](#).
- The sixth argument (`finished`) is a callback that will be executed when the animation has completed. This

callback takes two arguments, with the first argument indicating a final value, and the second argument being a `bool` that's set to `true` if the animation was canceled. Alternatively, the `finished` callback can be specified as an argument to the `Animation` constructor. However, with a single animation, if `finished` callbacks are specified in both the `Animation` constructor and the `Commit` method, only the callback specified in the `Commit` method will be executed.

- The seventh argument (`repeat`) is a callback that allows the animation to be repeated. It's called at the end of the animation, and returning `true` indicates that the animation should be repeated.

The overall effect is to create an animation that increases the `Scale` property of an `Image` from 1 to 2, over 2 seconds (2000 milliseconds), using the `Linear` easing function. Each time the animation completes, its `Scale` property is reset to 1 and the animation repeats.

#### NOTE

Concurrent animations, that run independently of each other can be constructed by creating an `Animation` object for each animation, and then calling the `Commit` method on each animation.

## Child Animations

The `Animation` class also supports child animations, which involves creating an `Animation` object to which other `Animation` objects are added. This enables a series of animations to be run and synchronized. The following code example demonstrates creating and running child animations:

```
var parentAnimation = new Animation ();
var scaleUpAnimation = new Animation (v => image.Scale = v, 1, 2, Easing.SpringIn);
var rotateAnimation = new Animation (v => image.Rotation = v, 0, 360);
var scaleDownAnimation = new Animation (v => image.Scale = v, 2, 1, Easing.SpringOut);

parentAnimation.Add (0, 0.5, scaleUpAnimation);
parentAnimation.Add (0, 1, rotateAnimation);
parentAnimation.Add (0.5, 1, scaleDownAnimation);

parentAnimation.Commit (this, "ChildAnimations", 16, 4000, null, (v, c) => SetIsEnabledButtonState (true, false));
```

Alternatively, the code example can be written more concisely, as demonstrated in the following code example:

```
new Animation {
    { 0, 0.5, new Animation (v => image.Scale = v, 1, 2) },
    { 0, 1, new Animation (v => image.Rotation = v, 0, 360) },
    { 0.5, 1, new Animation (v => image.Scale = v, 2, 1) }
}.Commit (this, "ChildAnimations", 16, 4000, null, (v, c) => SetIsEnabledButtonState (true, false));
```

In both code examples, a parent `Animation` object is created, to which additional `Animation` objects are then added. The first two arguments to the `Add` method specify when to begin and finish the child animation. The argument values must be between 0 and 1, and represent the relative period within the parent animation that the specified child animation will be active. Therefore, in this example the `scaleUpAnimation` will be active for the first half of the animation, the `scaleDownAnimation` will be active for the second half of the animation, and the `rotateAnimation` will be active for the entire duration.

The overall effect is that the animation occurs over 4 seconds (4000 milliseconds). The `scaleUpAnimation` animates the `Scale` property from 1 to 2, over 2 seconds. The `scaleDownAnimation` then animates the `Scale` property from 2 to 1, over 2 seconds. While both scale animations are occurring, the `rotateAnimation` animates the `Rotation` property from 0 to 360, over 4 seconds. Note that the scaling animations also use easing functions. The `SpringIn` easing function causes the `Image` to initially shrink before getting larger, and the `SpringOut` easing function

causes the `Image` to become smaller than its actual size towards the end of the complete animation.

There are a number of differences between an `Animation` object that uses child animations, and one that doesn't:

- When using child animations, the `finished` callback on a child animation indicates when the child has completed, and the `finished` callback passed to the `Commit` method indicates when the entire animation has completed.
- When using child animations, returning `true` from the `repeat` callback on the `Commit` method will not cause the animation to repeat, but the animation will continue to run without new values.
- When including an easing function in the `Commit` method, and the easing function returns a value greater than 1, the animation will be terminated. If the easing function returns a value less than 0, the value is clamped to 0. To use an easing function that returns a value less than 0 or greater than 1, it must be specified in one of the child animations, rather than in the `Commit` method.

The `Animation` class also includes `WithConcurrent` methods that can be used to add child animations to a parent `Animation` object. However, their `begin` and `finish` argument values aren't restricted to 0 to 1, but only that part of the child animation that corresponds to a range of 0 to 1 will be active. For example, if a `WithConcurrent` method call defines a child animation that targets a `Scale` property from 1 to 6, but with `begin` and `finish` values of -2 and 3, the `begin` value of -2 corresponds to a `Scale` value of 1, and the `finish` value of 3 corresponds to a `Scale` value of 6. Because values outside the range of 0 and 1 play no part in an animation, the `Scale` property will only be animated from 3 to 6.

## Cancelling an Animation

An application can cancel an animation with a call to the `AbortAnimation` extension method, as demonstrated in the following code example:

```
this.AbortAnimation ("SimpleAnimation");
```

Note that animations are uniquely identified by a combination of the animation owner, and the animation name. Therefore, the owner and name specified when running the animation must be specified to cancel the animation. Therefore, the code example will immediately cancel the animation named `simpleAnimation` that's owned by the page.

## Creating a Custom Animation

The examples shown here so far have demonstrated animations that could equally be achieved with the methods in the `ViewExtensions` class. However, the advantage of the `Animation` class is that it has access to the `callback` method, which is executed when the animated value changes. This allows the callback to implement any desired animation. For example, the following code example animates the `BackgroundColor` property of a page by setting it to `Color` values created by the `Color.FromHsla` method, with hue values ranging from 0 to 1:

```
new Animation (callback: v => BackgroundColor = Color.FromHsla (v, 1, 0.5),
    start: 0,
    end: 1).Commit (this, "Animation", 16, 4000, Easing.Linear, (v, c) => BackgroundColor = Color.Default);
```

The resulting animation provides the appearance of advancing the page background through the colors of the rainbow.

For more examples of creating complex animations, including a Bezier curve animation, see [Chapter 22](#) of [Creating Mobile Apps with Xamarin.Forms](#).

# Creating a Custom Animation Extension Method

The extension methods in the `ViewExtensions` class animate a property from its current value to a specified value. This makes it difficult to create, for example, a `ColorTo` animation method that can be used to animate a color from one value to another, because:

- The only `Color` property defined by the `VisualElement` class is `BackgroundColor`, which isn't always the desired `Color` property to animate.
- Often the current value of a `Color` property is `Color.Default`, which isn't a real color, and which can't be used in interpolation calculations.

The solution to this problem is to not have the `ColorTo` method target a particular `Color` property. Instead, it can be written with a callback method that passes the interpolated `Color` value back to the caller. In addition, the method will take start and end `Color` arguments.

The `ColorTo` method can be implemented as an extension method that uses the `Animate` method in the `AnimationExtensions` class to provide its functionality. This is because the `Animate` method can be used to target properties that aren't of type `double`, as demonstrated in the following code example:

```
public static class ViewExtensions
{
    public static Task<bool> ColorTo(this VisualElement self, Color fromColor, Color toColor, Action<Color>
callback, uint length = 250, Easing easing = null)
    {
        Func<double, Color> transform = (t) =>
            Color.FromRgba(fromColor.R + t * (toColor.R - fromColor.R),
                           fromColor.G + t * (toColor.G - fromColor.G),
                           fromColor.B + t * (toColor.B - fromColor.B),
                           fromColor.A + t * (toColor.A - fromColor.A));
        return ColorAnimation(self, "ColorTo", transform, callback, length, easing);
    }

    public static void CancelAnimation(this VisualElement self)
    {
        self.AbortAnimation("ColorTo");
    }

    static Task<bool> ColorAnimation(VisualElement element, string name, Func<double, Color> transform,
Action<Color> callback, uint length, Easing easing)
    {
        easing = easing ?? Easing.Linear;
        var taskCompletionSource = new TaskCompletionSource<bool>();

        element.Animate<Color>(name, transform, callback, 16, length, easing, (v, c) =>
taskCompletionSource.SetResult(c));
        return taskCompletionSource.Task;
    }
}
```

The `Animate` method requires a *transform* argument, which is a callback method. The input to this callback is always a `double` ranging from 0 to 1. Therefore, the `ColorTo` method defines its own transform `Func` that accepts a `double` ranging from 0 to 1, and that returns a `Color` value corresponding to that value. The `Color` value is calculated by interpolating the `R`, `G`, `B`, and `A` values of the two supplied `Color` arguments. The `Color` value is then passed to the callback method for application to a particular property.

This approach allows the `ColorTo` method to animate any `Color` property, as demonstrated in the following code example:

```
await Task.WhenAll(
    label.ColorTo(Color.Red, Color.Blue, c => label.TextColor = c, 5000),
    label.ColorTo(Color.Blue, Color.Red, c => label.BackgroundColor = c, 5000));
await this.ColorTo(Color.FromRgb(0, 0, 0), Color.FromRgb(255, 255, 255), c => BackgroundColor = c, 5000);
await boxView.ColorTo(Color.Blue, Color.Red, c => boxView.Color = c, 4000);
```

In this code example, the `ColorTo` method animates the `TextColor` and `BackgroundColor` properties of a `Label`, the `BackgroundColor` property of a page, and the `Color` property of a `BoxView`.

## Summary

This article demonstrated how to use the `Animation` class to create and cancel animations, synchronize multiple animations, and create custom animations that animate properties that aren't animated by the existing animation methods. The `Animation` class is the building block of all Xamarin.Forms animations.

## Related Links

- [Custom Animations \(sample\)](#)
- [Animation](#)
- [AnimationExtensions](#)

# Xamarin.Forms BoxView

9/20/2018 • 18 minutes to read • [Edit Online](#)

`BoxView` renders a simple rectangle of a specified width, height, and color. You can use `BoxView` for decoration, rudimentary graphics, and for interaction with the user through touch.

Because Xamarin.Forms does not have a built-in vector graphics system, the `BoxView` helps to compensate. Some of the sample programs described in this article use `BoxView` for rendering graphics. The `BoxView` can be sized to resemble a line of a specific width and thickness, and then rotated by any angle using the `Rotation` property.

Although `BoxView` can mimic simple graphics, you might want to investigate [Using SkiaSharp in Xamarin.Forms](#) for more sophisticated graphics requirements.

This article discusses the following topics:

- [Setting BoxView Color and Size](#) – set the `BoxView` properties.
- [Rendering Text Decorations](#) – use a `BoxView` for rendering lines.
- [Listing Colors with BoxView](#) – display all the system colors in a `ListView`.
- [Playing the Game of Life by Subclassing BoxView](#) – implement a famous cellular automaton.
- [Creating a Digital Clock](#) – simulate a dot-matrix display.
- [Creating an Analog Clock](#) – transform and animate `BoxView` elements.

## Setting BoxView Color and Size

Typically you'll set the following properties of `BoxView`:

- `Color` to set its color.
- `CornerRadius` to set its corner radius.
- `WidthRequest` to set the width of the `BoxView` in device-independent units.
- `HeightRequest` to set the height of the `BoxView`.

The `Color` property is of type `Color`; the property can be set to any `Color` value, including the 141 static read-only fields of named colors ranging alphabetically from `AliceBlue` to `YellowGreen`.

The `CornerRadius` property is of type `CornerRadius`; the property can be set to a single `double` uniform corner radius value, or a `CornerRadius` structure defined by four `double` values that are applied to the top left, top right, bottom left, and bottom right of the `BoxView`.

The `WidthRequest` and `HeightRequest` properties only play a role if the `BoxView` is *unconstrained* in layout. This is the case when the layout container needs to know the child's size, for example, when the `BoxView` is a child of an auto-sized cell in the `Grid` layout. A `BoxView` is also unconstrained when its `HorizontalOptions` and `VerticalOptions` properties are set to values other than `LayoutOptions.Fill`. If the `BoxView` is unconstrained, but the `WidthRequest` and `HeightRequest` properties are not set, then the width or height are set to default values of 40 units, or about 1/4 inch on mobile devices.

The `WidthRequest` and `HeightRequest` properties are ignored if the `BoxView` is *constrained* in layout, in which case the layout container imposes its own size on the `BoxView`.

A `BoxView` can be constrained in one dimension and unconstrained in the other. For example, if the `BoxView` is a child of a vertical `StackLayout`, the vertical dimension of the `BoxView` is unconstrained and its horizontal dimension is generally constrained. But there are exceptions for that horizontal dimension: If the `BoxView` has its

`HorizontalOptions` property set to something other than `LayoutOptions.Fill`, then the horizontal dimension is also unconstrained. It's also possible for the `StackLayout` itself to have an unconstrained horizontal dimension, in which case the `BoxView` will also be horizontally unconstrained.

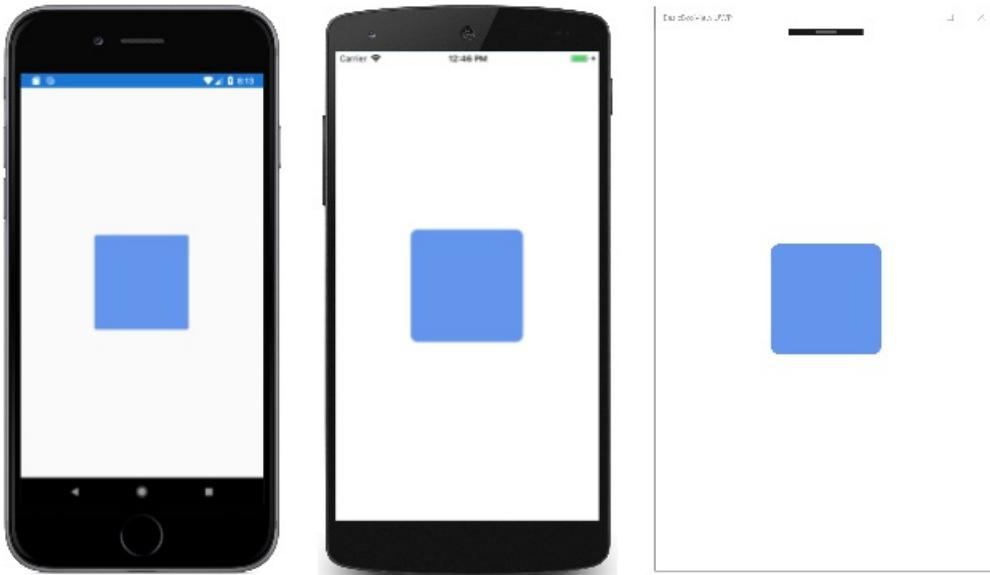
The [BasicBoxView](#) sample displays a one-inch-square unconstrained `BoxView` in the center of its page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:BasicBoxView"
    x:Class="BasicBoxView.MainPage">

    <BoxView Color="CornflowerBlue"
        CornerRadius="10"
        WidthRequest="160"
        HeightRequest="160"
        VerticalOptions="Center"
        HorizontalOptions="Center" />

</ContentPage>
```

Here's the result:



If the `VerticalOptions` and `HorizontalOptions` properties are removed from the `BoxView` tag or are set to `Fill`, then the `BoxView` becomes constrained by the size of the page, and expands to fill the page.

A `BoxView` can also be a child of an `AbsoluteLayout`. In that case, both the location and size of the `BoxView` are set using the `LayoutBounds` attached bindable property. The `AbsoluteLayout` is discussed in the article [AbsoluteLayout](#).

You'll see examples of all these cases in the sample programs that follow.

## Rendering Text Decorations

You can use the `BoxView` to add some simple decorations on your pages in the form of horizontal and vertical lines. The [TextDecoration](#) sample demonstrates this. All of the program's visuals are defined in the `MainPage.xaml` file, which contains several `Label` and `BoxView` elements in the `StackLayout` shown here:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:TextDecoration"
    x:Class="TextDecoration.MainPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

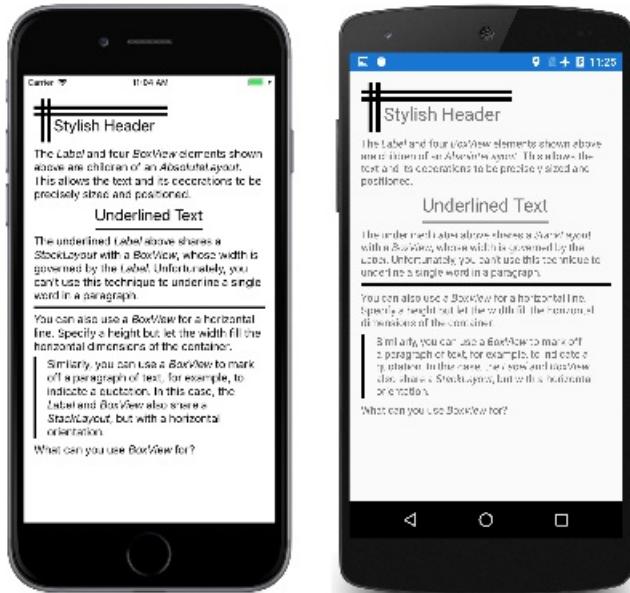
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="Color" Value="Black" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ScrollView Margin="15">
        <StackLayout>

        ...
        </StackLayout>
    </ScrollView>
</ContentPage>

```

All of the markup that follows are children of the `StackLayout`. This markup consists of several types of decorative `BoxView` elements used with the `Label` element:



The stylish header at the top of the page is achieved with an `AbsoluteLayout` whose children are four `BoxView` elements and a `Label`, all of which are assigned specific locations and sizes:

```

<AbsoluteLayout>
    <BoxView AbsoluteLayout.LayoutBounds="0, 10, 200, 5" />
    <BoxView AbsoluteLayout.LayoutBounds="0, 20, 200, 5" />
    <BoxView AbsoluteLayout.LayoutBounds="10, 0, 5, 65" />
    <BoxView AbsoluteLayout.LayoutBounds="20, 0, 5, 65" />
    <Label Text="Stylish Header"
        FontSize="24"
        AbsoluteLayout.LayoutBounds="30, 25, AutoSize, AutoSize"/>
</AbsoluteLayout>

```

In the XAML file, the `AbsoluteLayout` is followed by a `Label` with formatted text that describes the `AbsoluteLayout`.

You can underline a text string by enclosing both the `Label` and `BoxView` in a `StackLayout` that has its `HorizontalOptions` value set to something other than `Fill`. The width of the `StackLayout` is then governed by the width of the `Label`, which then imposes that width on the `BoxView`. The `BoxView` is assigned only an explicit height:

```
<StackLayout HorizontalOptions="Center">
    <Label Text="Underlined Text"
        FontSize="24" />
    <BoxView HeightRequest="2" />
</StackLayout>
```

You can't use this technique to underline individual words within longer text strings or a paragraph.

It's also possible to use a `BoxView` to resemble an HTML `hr` (horizontal rule) element. Simply let the width of the `BoxView` be determined by its parent container, which in this case is the `StackLayout`:

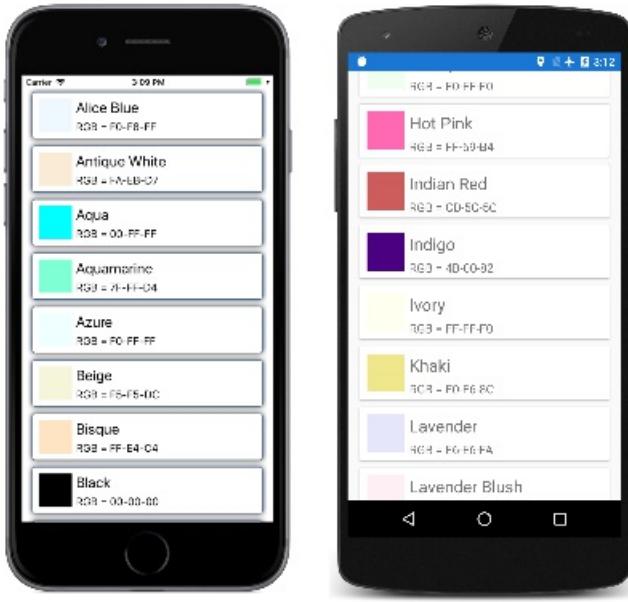
```
<BoxView HeightRequest="3" />
```

Finally, you can draw a vertical line on one side of a paragraph of text by enclosing both the `BoxView` and the `Label` in a horizontal `StackLayout`. In this case, the height of the `BoxView` is the same as the height of `StackLayout`, which is governed by the height of the `Label`:

```
<StackLayout Orientation="Horizontal">
    <BoxView WidthRequest="4"
        Margin="0, 0, 10, 0" />
    <Label>
        ...
    </Label>
</StackLayout>
```

## Listing Colors with BoxView

The `BoxView` is convenient for displaying colors. This program uses a `ListView` to list all the public static read-only fields of the `Xamarin.Forms` `Color` structure:



The **ListViewColors** program includes a class named `NamedColor`. The static constructor uses reflection to access all the fields of the `Color` structure and create a `NamedColor` object for each one. These are stored in the static `All` property:

```
public class NamedColor
{
    // Instance members.
    private NamedColor()
    {
    }

    public string Name { private set; get; }

    public string FriendlyName { private set; get; }

    public Color Color { private set; get; }

    public string RgbDisplay { private set; get; }

    // Static members.
    static NamedColor()
    {
        List<NamedColor> all = new List<NamedColor>();
        StringBuilder stringBuilder = new StringBuilder();

        // Loop through the public static fields of the Color structure.
        foreach (FieldInfo fieldInfo in typeof(Color).GetRuntimeFields ())
        {
            if (fieldInfo.IsPublic &&
                fieldInfo.IsStatic &&
                fieldInfo.FieldType == typeof (Color))
            {
                // Convert the name to a friendly name.
                string name = fieldInfo.Name;
                stringBuilder.Clear();
                int index = 0;

                foreach (char ch in name)
                {
                    if (index != 0 && Char.IsUpper(ch))
                    {
                        stringBuilder.Append(' ');
                    }
                    stringBuilder.Append(ch);
                    index++;
                }
            }
        }
    }
}
```

```
// Instantiate a NamedColor object.  
Color color = (Color)fieldInfo.GetValue(null);  
  
NamedColor namedColor = new NamedColor  
{  
    Name = name,  
    FriendlyName = stringBuilder.ToString(),  
    Color = color,  
    RgbDisplay = String.Format("{0:X2}-{1:X2}-{2:X2}",  
        (int)(255 * color.R),  
        (int)(255 * color.G),  
        (int)(255 * color.B))  
};  
  
// Add it to the collection.  
all.Add(namedColor);  
}  
}  
all.TrimExcess();  
All = all;  
}  
  
public static IList<NamedColor> All { private set; get; }  
}
```

The program visuals are described in the XAML file. The `ItemsSource` property of the `ListView` is set to the static `NamedColor.All` property, which means that the `ListView` displays all the individual `NamedColor` objects:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ListViewColors"
    x:Class="ListViewColors.MainPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="10, 20, 10, 0" />
            <On Platform="Android, UWP" Value="10, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <ListView SeparatorVisibility="None">
        <ItemsSource="{x:Static local:NamedColor.All}">
            <ListView.RowHeight>
                <OnPlatform x:TypeArguments="x:Int32">
                    <On Platform="iOS, Android" Value="80" />
                    <On Platform="UWP" Value="90" />
                </OnPlatform>
            </ListView.RowHeight>

            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ContentView Padding="5">
                            <Frame OutlineColor="Accent"
                                Padding="10">
                                <StackLayout Orientation="Horizontal">
                                    <BoxView Color="{Binding Color}"
                                        WidthRequest="50"
                                        HeightRequest="50" />
                                <StackLayout>
                                    <Label Text="{Binding FriendlyName}"
                                        FontSize="22"
                                        VerticalOptions="StartAndExpand" />
                                    <Label Text="{Binding RgbDisplay, StringFormat='RGB = {0}'}"
                                        FontSize="16"
                                        VerticalOptions="CenterAndExpand" />
                                </StackLayout>
                            </StackLayout>
                        </Frame>
                    </ContentView>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>

```

The `NamedColor` objects are formatted by the `ViewCell` object that is set as the data template of the `ListView`. This template includes a `BoxView` whose `Color` property is bound to the `Color` property of the `NamedColor` object.

## Playing the Game of Life by Subclassing BoxView

The Game of Life is a cellular automaton invented by mathematician John Conway and popularized in the pages of *Scientific American* in the 1970s. A good introduction is provided by the Wikipedia article [Conway's Game of Life](#).

The Xamarin.Forms **GameOfLife** program defines a class named `LifeCell` that derives from `BoxView`. This class encapsulates the logic of an individual cell in the Game of Life:

```

class LifeCell : BoxView
{
    bool isAlive;

    public event EventHandler Tapped;

    public LifeCell()
    {
        BackgroundColor = Color.White;

        TapGestureRecognizer tapGesture = new TapGestureRecognizer();
        tapGesture.Tapped += (sender, args) =>
        {
            Tapped?.Invoke(this, EventArgs.Empty);
        };
        GestureRecognizers.Add(tapGesture);
    }

    public int Col { set; get; }

    public int Row { set; get; }

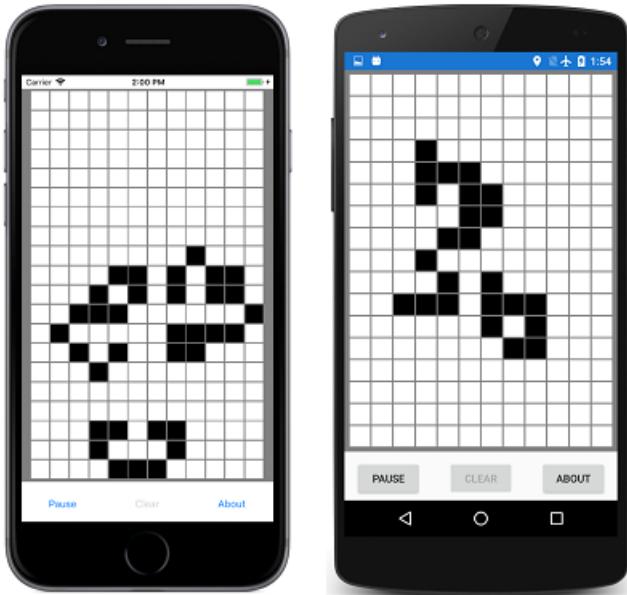
    public bool IsAlive
    {
        set
        {
            if (isAlive != value)
            {
                isAlive = value;
                BackgroundColor = isAlive ? Color.Black : Color.White;
            }
        }
        get
        {
            return isAlive;
        }
    }
}

```

`LifeCell` adds three more properties to `BoxView`: the `Col` and `Row` properties store the position of the cell within the grid, and the `IsAlive` property indicates its state. The `IsAlive` property also sets the `color` property of the `BoxView` to black if the cell is alive, and white if the cell is not alive.

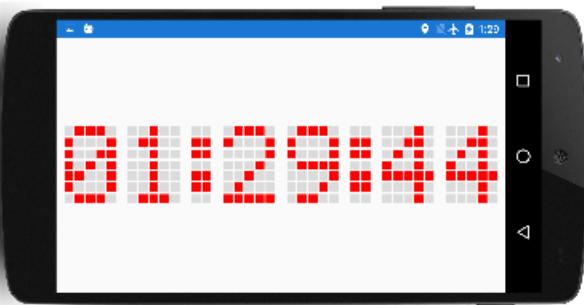
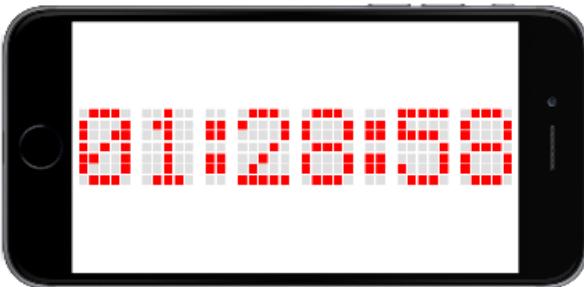
`LifeCell` also installs a `TapGestureRecognizer` to allow the user to toggle the state of cells by tapping them. The class translates the `Tapped` event from the gesture recognizer into its own `Tapped` event.

The `GameOfLife` program also includes a `LifeGrid` class that encapsulates much of the logic of the game, and a `MainPage` class that handles the program's visuals. These include an overlay that describes the rules of the game. Here is the program in action showing a couple hundred `LifeCell` objects on the page:



## Creating a Digital Clock

The **DotMatrixClock** program creates 210 `BoxView` elements to simulate the dots of an old-fashioned 5-by-7 dot-matrix display. You can read the time in either portrait or landscape mode, but it's larger in landscape:



The XAML file does little more than instantiate the `AbsoluteLayout` used for the clock:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DotMatrixClock"
    x:Class="DotMatrixClock.MainPage"
    Padding="10"
    SizeChanged="OnPageSizeChanged">

    <AbsoluteLayout x:Name="absoluteLayout"
        VerticalOptions="Center" />
</ContentPage>
```

Everything else occurs in the code-behind file. The dot-matrix display logic is greatly simplified by the definition of several arrays that describe the dots corresponding to each of the 10 digits and a colon:

```

public partial class MainPage : ContentPage
{
    // Total dots horizontally and vertically.
    const int horzDots = 41;
    const int vertDots = 7;

    // 5 x 7 dot matrix patterns for 0 through 9.
    static readonly int[, ] numberPatterns = new int[10, 7, 5]
    {
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 1, 1}, { 1, 0, 1, 0, 1},
            { 1, 1, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 0, 0}, { 0, 1, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0},
            { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0},
            { 0, 0, 1, 0, 0}, { 0, 1, 0, 0, 0}, { 1, 1, 1, 1, 1}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 0, 1, 0},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 0, 1, 0}, { 0, 0, 1, 1, 0}, { 0, 1, 0, 1, 0}, { 1, 0, 0, 1, 0},
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 0, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0}, { 0, 0, 0, 0, 1},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 1, 0}, { 0, 1, 0, 0, 0}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0},
            { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 1},
            { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 1, 1, 0, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 1},
            { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 1, 1, 0, 0}
        }
    };
}

// Dot matrix pattern for a colon.
static readonly int[,] colonPattern = new int[7, 2]
{
    { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }
};

// BoxView colors for on and off.
static readonly Color colorOn = Color.Red;
static readonly Color colorOff = new Color(0.5, 0.5, 0.5, 0.25);

// Box views for 6 digits, 7 rows, 5 columns.
BoxView[, , ] digitBoxViews = new BoxView[6, 7, 5];

...
}

```

These fields conclude with a three-dimensional array of `BoxView` elements for storing the dot patterns for the six digits.

The constructor creates all the `BoxView` elements for the digits and colon, and also initializes the `color` property of the `BoxView` elements for the colon:

```
public partial class MainPage : ContentPage
{
    ...

    public MainPage()
    {
        InitializeComponent();

        // BoxView dot dimensions.
        double height = 0.85 / vertDots;
        double width = 0.85 / horzDots;

        // Create and assemble the BoxViews.
        double xIncrement = 1.0 / (horzDots - 1);
        double yIncrement = 1.0 / (vertDots - 1);
        double x = 0;

        for (int digit = 0; digit < 6; digit++)
        {
            for (int col = 0; col < 5; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the digit BoxView and add to layout.
                    BoxView boxView = new BoxView();
                    digitBoxViews[digit, row, col] = boxView;
                    absoluteLayout.Children.Add(boxView,
                        new Rectangle(x, y, width, height),
                        AbsoluteLayoutFlags.All);

                    y += yIncrement;
                }
                x += xIncrement;
            }
            x += xIncrement;
        }

        // Colons between the hours, minutes, and seconds.
        if (digit == 1 || digit == 3)
        {
            int colon = digit / 2;

            for (int col = 0; col < 2; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the BoxView and set the color.
                    BoxView boxView = new BoxView
                    {
                        Color = colonPattern[row, col] == 1 ?
                            colorOn : colorOff
                    };
                    absoluteLayout.Children.Add(boxView,
                        new Rectangle(x, y, width, height),
                        AbsoluteLayoutFlags.All);

                    y += yIncrement;
                }
            }
        }
    }
}
```

```

        x += xIncrement;
    }
    x += xIncrement;
}
}

// Set the timer and initialize with a manual call.
Device.StartTimer(TimeSpan.FromSeconds(1), OnTimer);
OnTimer();
}

...
}

}

```

This program uses the relative positioning and sizing feature of `AbsoluteLayout`. The width and height of each `BoxView` are set to fractional values, specifically 85% of 1 divided by the number of horizontal and vertical dots. The positions are also set to fractional values.

Because all the positions and sizes are relative to the total size of the `AbsoluteLayout`, the `SizeChanged` handler for the page need only set a `HeightRequest` of the `AbsoluteLayout`:

```

public partial class MainPage : ContentPage
{
    ...

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // No chance a display will have an aspect ratio > 41:7
        absoluteLayout.HeightRequest = vertDots * Width / horzDots;
    }

    ...
}

```

The width of the `AbsoluteLayout` is automatically set because it stretches to the full width of the page.

The final code in the `MainPage` class processes the timer callback and colors the dots of each digit. The definition of the multi-dimensional arrays at the beginning of the code-behind file helps make this logic the simplest part of the program:

```

public partial class MainPage : ContentPage
{
    ...

    bool OnTimer()
    {
        DateTime dateTime = DateTime.Now;

        // Convert 24-hour clock to 12-hour clock.
        int hour = (dateTime.Hour + 11) % 12 + 1;

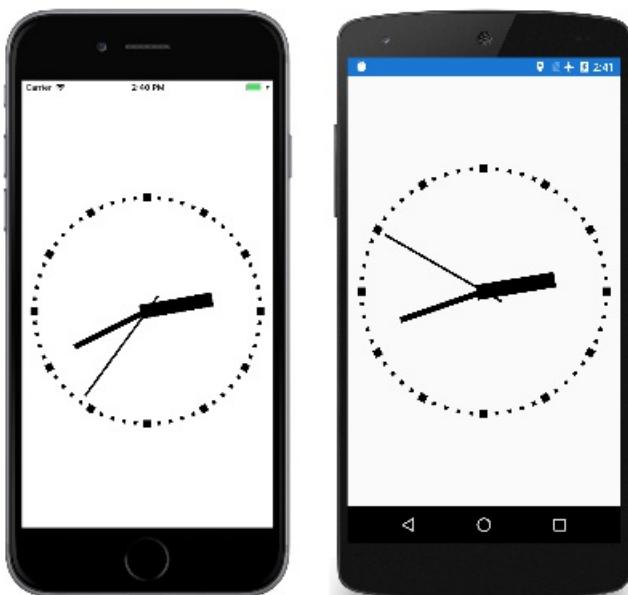
        // Set the dot colors for each digit separately.
        SetDotMatrix(0, hour / 10);
        SetDotMatrix(1, hour % 10);
        SetDotMatrix(2, dateTime.Minute / 10);
        SetDotMatrix(3, dateTime.Minute % 10);
        SetDotMatrix(4, dateTime.Second / 10);
        SetDotMatrix(5, dateTime.Second % 10);
        return true;
    }

    void SetDotMatrix(int index, int digit)
    {
        for (int row = 0; row < 7; row++)
            for (int col = 0; col < 5; col++)
            {
                bool isOn = numberPatterns[digit, row, col] == 1;
                Color color = isOn ? colorOn : colorOff;
                digitBoxViews[index, row, col].Color = color;
            }
    }
}

```

## Creating an Analog Clock

A dot-matrix clock might seem to be an obvious application of `BoxView`, but `BoxView` elements are also capable of realizing an analog clock:



All the visuals in the **BoxViewClock** program are children of an `AbsoluteLayout`. These elements are sized using the `LayoutBounds` attached property, and rotated using the `Rotation` property.

The three `BoxView` elements for the hands of the clock are instantiated in the XAML file, but not positioned or

sized:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:BoxViewClock"
    x:Class="BoxViewClock.MainPage">
<ContentPage.Padding>
    <OnPlatform x:TypeArguments="Thickness">
        <On Platform="iOS" Value="0, 20, 0, 0" />
    </OnPlatform>
</ContentPage.Padding>

<AbsoluteLayout x:Name="absoluteLayout"
    SizeChanged="OnAbsoluteLayoutSizeChanged">

    <BoxView x:Name="hourHand"
        Color="Black" />

    <BoxView x:Name="minuteHand"
        Color="Black" />

    <BoxView x:Name="secondHand"
        Color="Black" />
</AbsoluteLayout>
</ContentPage>
```

The constructor of the code-behind file instantiates the 60 `BoxView` elements for the tick marks around the circumference of the clock:

```
public partial class MainPage : ContentPage
{
    ...

    BoxView[] tickMarks = new BoxView[60];

    public MainPage()
    {
        InitializeComponent();

        // Create the tick marks (to be sized and positioned later).
        for (int i = 0; i < tickMarks.Length; i++)
        {
            tickMarks[i] = new BoxView { Color = Color.Black };
            absoluteLayout.Children.Add(tickMarks[i]);
        }

        Device.StartTimer(TimeSpan.FromSeconds(1.0 / 60), OnTimerTick);
    }

    ...
}
```

The sizing and positioning of all the `BoxView` elements occurs in the `SizeChanged` handler for the `AbsoluteLayout`. A little structure internal to the class called `HandParams` describes the size of each of the three hands relative to the total size of the clock:

```

public partial class MainPage : ContentPage
{
    // Structure for storing information about the three hands.
    struct HandParams
    {
        public HandParams(double width, double height, double offset) : this()
        {
            Width = width;
            Height = height;
            Offset = offset;
        }

        public double Width { private set; get; } // fraction of radius
        public double Height { private set; get; } // ditto
        public double Offset { private set; get; } // relative to center pivot
    }

    static readonly HandParams secondParams = new HandParams(0.02, 1.1, 0.85);
    static readonly HandParams minuteParams = new HandParams(0.05, 0.8, 0.9);
    static readonly HandParams hourParams = new HandParams(0.125, 0.65, 0.9);

    ...
}

```

The `SizeChanged` handler determines the center and radius of the `AbsoluteLayout`, and then sizes and positions the 60 `BoxView` elements used as tick marks. The `for` loop concludes by setting the `Rotation` property of each of these `BoxView` elements. At the end of the `SizeChanged` handler, the `LayoutHand` method is called to size and position the three hands of the clock:

```

public partial class MainPage : ContentPage
{
    ...

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        // Get the center and radius of the AbsoluteLayout.
        Point center = new Point(AbsoluteLayout.Width / 2, AbsoluteLayout.Height / 2);
        double radius = 0.45 * Math.Min(AbsoluteLayout.Width, AbsoluteLayout.Height);

        // Position, size, and rotate the 60 tick marks.
        for (int index = 0; index < tickMarks.Length; index++)
        {
            double size = radius / (index % 5 == 0 ? 15 : 30);
            double radians = index * 2 * Math.PI / tickMarks.Length;
            double x = center.X + radius * Math.Sin(radians) - size / 2;
            double y = center.Y - radius * Math.Cos(radians) - size / 2;
            AbsoluteLayout.SetLayoutBounds(tickMarks[index], new Rectangle(x, y, size, size));
            tickMarks[index].Rotation = 180 * radians / Math.PI;
        }

        // Position and size the three hands.
        LayoutHand(secondHand, secondParams, center, radius);
        LayoutHand(minuteHand, minuteParams, center, radius);
        LayoutHand(hourHand, hourParams, center, radius);
    }

    void LayoutHand(BoxView boxView, HandParams handParams, Point center, double radius)
    {
        double width = handParams.Width * radius;
        double height = handParams.Height * radius;
        double offset = handParams.Offset;

        AbsoluteLayout.SetLayoutBounds(boxView,
            new Rectangle(center.X - 0.5 * width,
                center.Y - offset * height,
                width, height));

        // Set the AnchorY property for rotations.
        boxView.AnchorY = handParams.Offset;
    }

    ...
}

```

The `LayoutHand` method sizes and positions each hand to point straight up to the 12:00 position. At the end of the method, the `AnchorY` property is set to a position corresponding to the center of the clock. This indicates the center of rotation.

The hands are rotated in the timer callback function:

```

public partial class MainPage : ContentPage
{
    ...

    bool OnTimerTick()
    {
        // Set rotation angles for hour and minute hands.
        DateTime dateTime = DateTime.Now;
        hourHand.Rotation = 30 * (dateTime.Hour % 12) + 0.5 * dateTime.Minute;
        minuteHand.Rotation = 6 * dateTime.Minute + 0.1 * dateTime.Second;

        // Do an animation for the second hand.
        double t = dateTime.Millisecond / 1000.0;

        if (t < 0.5)
        {
            t = 0.5 * Easing.SpringIn.Ease(t / 0.5);
        }
        else
        {
            t = 0.5 * (1 + Easing.SpringOut.Ease((t - 0.5) / 0.5));
        }

        secondHand.Rotation = 6 * (dateTime.Second + t);
        return true;
    }
}

```

The second hand is treated a little differently: An animation easing function is applied to make the movement seem mechanical rather than smooth. On each tick, the second hand pulls back a little and then overshoots its destination. This little bit of code adds a lot to the realism of the movement.

## Conclusion

The `BoxView` might seem simple at first, but as you've seen, it can be quite versatile, and can almost reproduce visuals that are normally possible only with vector graphics. For more sophisticated graphics, consult [Using SkiaSharp in Xamarin.Forms](#).

## Related Links

- [Basic BoxView \(sample\)](#)
- [Text Decoration \(sample\)](#)
- [Color ListBox \(sample\)](#)
- [Game of Life \(sample\)](#)
- [Dot-Matrix Clock \(sample\)](#)
- [BoxView Clock \(sample\)](#)
- [BoxView](#)

# Xamarin.Forms Button

11/20/2018 • 19 minutes to read • [Edit Online](#)

The `Button` responds to a tap or click that directs an application to carry out a particular task.

The `Button` is the most fundamental interactive control in all of Xamarin.Forms. The `Button` usually displays a short text string indicating a command, but it can also display a bitmap image, or a combination of text and an image. The user presses the `Button` with a finger or clicks it with a mouse to initiate that command.

Most of the topics discussed below correspond to pages in the [ButtonDemos](#) sample.

## Handling button clicks

`Button` defines a `Clicked` event that is fired when the user taps the `Button` with a finger or mouse pointer. The event is fired when the finger or mouse button is released from the surface of the `Button`. The `Button` must have its `.IsEnabled` property set to `true` for it to respond to taps.

The **Basic Button Click** page in the [ButtonDemos](#) sample demonstrates how to instantiate a `Button` in XAML and handle its `Clicked` event. The `BasicButtonClickPage.xaml` file contains a `StackLayout` with both a `Label` and a `Button`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ButtonDemos.BasicButtonClickPage"
    Title="Basic Button Click">
    <StackLayout>

        <Label x:Name="label"
            Text="Click the Button below"
            FontSize="Large"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center" />

        <Button Text="Click to Rotate Text!"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center"
            Clicked="OnButtonClicked" />

    </StackLayout>
</ContentPage>
```

The `Button` tends to occupy all the space that's allowed for it. For example, if you don't set the `HorizontalOptions` property of `Button` to something other than `Fill`, the `Button` will occupy the full width of its parent.

By default, the `Button` is rectangular, but you can give it rounded corners by using the `CornerRadius` property, as described below in the section [Button appearance](#).

The `Text` property specifies the text that appears in the `Button`. The `Clicked` event is set to an event handler named `OnButtonClicked`. This handler is located in the code-behind file, `BasicButtonClickPage.xaml.cs`:

```

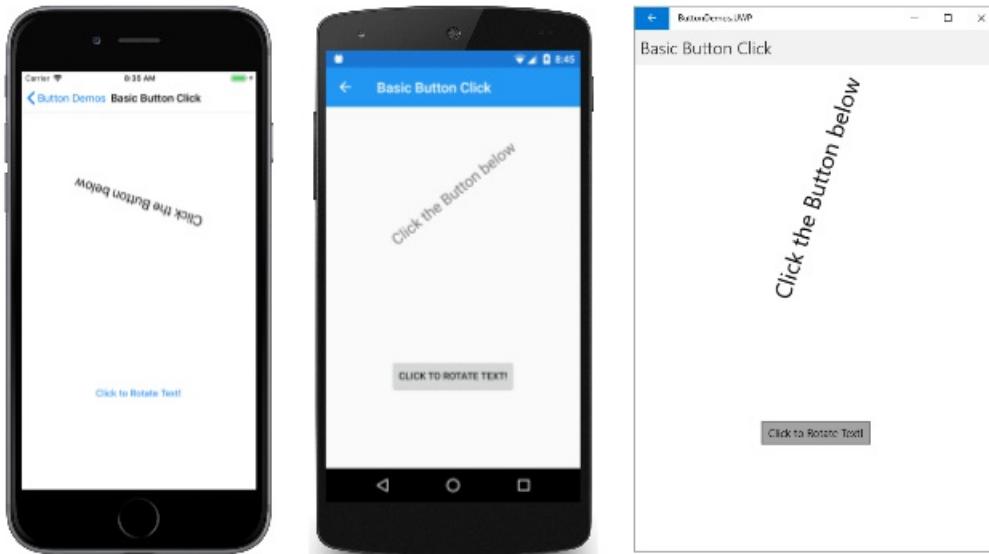
public partial class BasicButtonClickPage : ContentPage
{
    public BasicButtonClickPage ()
    {
        InitializeComponent ();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        await label.RelRotateTo(360, 1000);
    }
}

```

When the `Button` is tapped, the `OnButtonClicked` method executes. The `sender` argument is the `Button` object responsible for this event. You can use this to access the `Button` object, or to distinguish between multiple `Button` objects sharing the same `clicked` event.

This particular `clicked` handler calls an animation function that rotates the `Label` 360 degrees in 1000 milliseconds. Here's the program running on iOS and Android devices, and as a Universal Windows Platform (UWP) application on the Windows 10 desktop:



Notice that the `onButtonClicked` method includes the `async` modifier because `await` is used within the event handler. A `clicked` event handler requires the `async` modifier only if the body of the handler uses `await`.

Each platform renders the `Button` in its own specific manner. In the **Button appearance** section, you'll see how to set colors and make the `Button` border visible for more customized appearances. `Button` implements the `IFontElement` interface, so it includes `FontFamily`, `FontSize`, and `FontAttributes` properties.

## Creating a button in code

It's common to instantiate a `Button` in XAML, but you can also create a `Button` in code. This might be convenient when your application needs to create multiple buttons based on data that is enumerable with a `foreach` loop.

The **Code Button Click** page demonstrates how to create a page that is functionally equivalent to the **Basic Button Click** page but entirely in C#:

```

public class CodeButtonClickPage : ContentPage
{
    public CodeButtonClickPage ()
    {
        Title = "Code Button Click";

        Label label = new Label
        {
            Text = "Click the Button below",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        Button button = new Button
        {
            Text = "Click to Rotate Text!",
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };
        button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);

        Content = new StackLayout
        {
            Children =
            {
                label,
                button
            }
        };
    }
}

```

Everything is done in the class's constructor. Because the `Clicked` handler is only one statement long, it can be attached to the event very simply:

```
button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);
```

Of course, you can also define the event handler as a separate method (just like the `OnButtonClick` method in [Basic Button Click](#)) and attach that method to the event:

```
button.Clicked += OnButtonClicked;
```

## Disabling the button

Sometimes an application is in a particular state where a particular `Button` click is not a valid operation. In those cases, the `Button` should be disabled by setting its `.IsEnabled` property to `false`. The classic example is an `Entry` control for a filename accompanied by a file-open `Button`: The `Button` should be enabled only if some text has been typed into the `Entry`. You can use a `DataTrigger` for this task, as shown in the [Data Triggers](#) article.

## Using the command interface

It is possible for an application to respond to `Button` taps without handling the `Clicked` event. The `Button` implements an alternative notification mechanism called the *command* or *commanding* interface. This consists of two properties:

- `Command` of type `ICommand`, an interface defined in the `System.Windows.Input` namespace.

- `CommandParameter` property of type `Object`.

This approach is particularly suitable in connection with data-binding, and particularly when implementing the Model-View-ViewModel (MVVM) architecture. These topics are discussed in the articles [Data Binding](#), [From Data Bindings to MVVM](#), and [MVVM](#).

In an MVVM application, the ViewModel defines properties of type `ICommand` that are then connected to the XAML `Button` elements with data bindings. Xamarin.Forms also defines `Command` and `Command<T>` classes that implement the `ICommand` interface and assist the ViewModel in defining properties of type `ICommand`.

Commanding is described in greater detail in the article [The Command Interface](#) but the [Basic Button Command](#) page in the [ButtonDemos](#) sample shows the basic approach.

The `CommandDemoViewModel` class is a very simple ViewModel that defines a property of type `double` named `Number`, and two properties of type `ICommand` named `MultiplyBy2Command` and `DivideBy2Command`:

```
class CommandDemoViewModel : INotifyPropertyChanged
{
    double number = 1;

    public event PropertyChangedEventHandler PropertyChanged;

    public CommandDemoViewModel()
    {
        MultiplyBy2Command = new Command(() => Number *= 2);

        DivideBy2Command = new Command(() => Number /= 2);
    }

    public double Number
    {
        set
        {
            if (number != value)
            {
                number = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Number"));
            }
        }
        get
        {
            return number;
        }
    }

    public ICommand MultiplyBy2Command { private set; get; }

    public ICommand DivideBy2Command { private set; get; }
}
```

The two `ICommand` properties are initialized in the class's constructor with two objects of type `Command`. The `Command` constructors include a little function (called the `execute` constructor argument) that either doubles or halves the `Number` property.

The `BasicButtonCommand.xaml` file sets its `BindingContext` to an instance of `CommandDemoViewModel`. The `Label` element and two `Button` elements contain bindings to the three properties in `CommandDemoViewModel`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ButtonDemos"
    x:Class="ButtonDemos.BasicButtonCommandPage"
    Title="Basic Button Command">

    <ContentPage.BindingContext>
        <local:CommandDemoViewModel />
    </ContentPage.BindingContext>

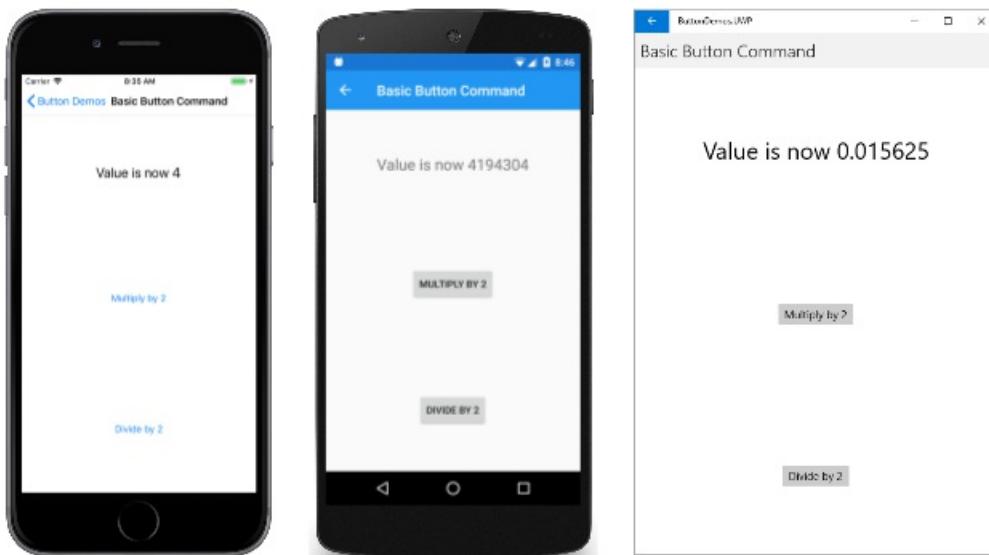
    <StackLayout>
        <Label Text="{Binding Number, StringFormat='Value is now {0}'}"
            FontSize="Large"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center" />

        <Button Text="Multiply by 2"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center"
            Command="{Binding MultiplyBy2Command}" />

        <Button Text="Divide by 2"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center"
            Command="{Binding DivideBy2Command}" />
    </StackLayout>
</ContentPage>

```

As the two `Button` elements are tapped, the commands are executed, and the number changes value:



The advantage of this approach over `Clicked` handlers is that all the logic involving the functionality of this page is located in the ViewModel rather than the code-behind file, achieving a better separation of the user interface from the business logic.

It is also possible for the `command` objects to control the enabling and disabling of the `Button` elements. For example, suppose you want to limit the range of number values between  $2^{10}$  and  $2^{-10}$ . You can add another function to the constructor (called the `canExecute` argument) that returns `true` if the `Button` should be enabled. Here's the modification to the `CommandDemoViewModel` constructor:

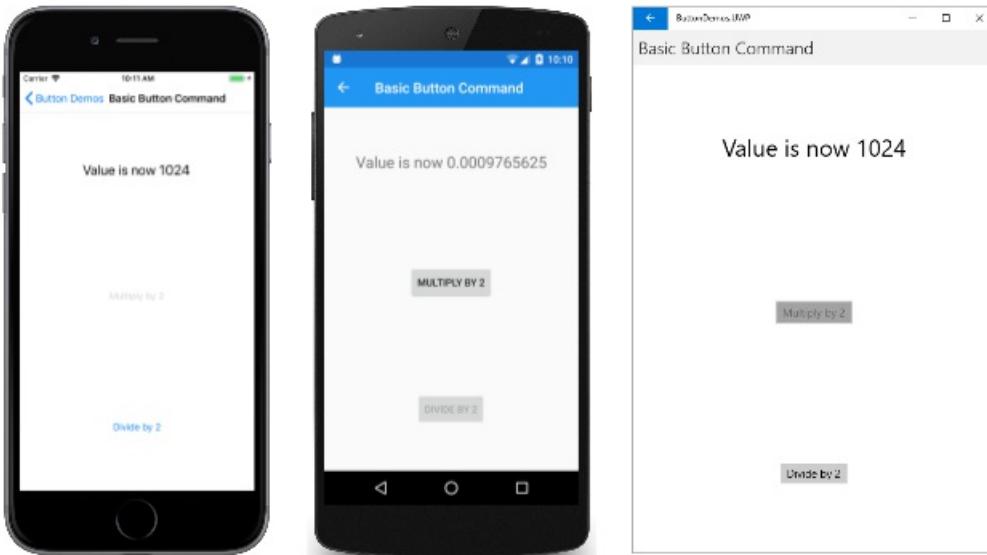
```

class CommandDemoViewModel : INotifyPropertyChanged
{
    ...
    public CommandDemoViewModel()
    {
        MultiplyBy2Command = new Command(
            execute: () =>
            {
                Number *= 2;
                ((Command)MultiplyBy2Command).ChangeCanExecute();
                ((Command)DivideBy2Command).ChangeCanExecute();
            },
            canExecute: () => Number < Math.Pow(2, 10));

        DivideBy2Command = new Command(
            execute: () =>
            {
                Number /= 2;
                ((Command)MultiplyBy2Command).ChangeCanExecute();
                ((Command)DivideBy2Command).ChangeCanExecute();
            },
            canExecute: () => Number > Math.Pow(2, -10));
    }
    ...
}

```

The calls to the `ChangeCanExecute` method of `Command` are necessary so that the `Command` method can call the `canExecute` method and determine whether the `Button` should be disabled or not. With this code change, as the number reaches the limit, the `Button` is disabled:



It is possible for two or more `Button` elements to be bound to the same `ICommand` property. The `Button` elements can be distinguished using the `CommandParameter` property of `Button`. In this case, you'll want to use the generic `Command<T>` class. The `CommandParameter` object is then passed as an argument to the `execute` and `canExecute` methods. This technique is shown in detail in the **Basic Commanding** section of the **Command Interface** article.

The **ButtonDemos** sample also uses this technique in its `MainPage` class. The `MainPage.xaml` file contains a `Button` for each page of the sample:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ButtonDemos"
    x:Class="ButtonDemos.MainPage"
    Title="Button Demos">

    <ScrollView>
        <FlexLayout Direction="Column"
            JustifyContent="SpaceEvenly"
            AlignItems="Center">

            <Button Text="Basic Button Click"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:BasicButtonClickPage}" />

            <Button Text="Code Button Click"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:CodeButtonClickPage}" />

            <Button Text="Basic Button Command"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:BasicButtonCommandPage}" />

            <Button Text="Press and Release Button"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:PressAndReleaseButtonPage}" />

            <Button Text="Button Appearance"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:ButtonAppearancePage}" />

            <Button Text="Toggle Button Demo"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:ToggleButtonDemoPage}" />

            <Button Text="Image Button Demo"
                Command="{Binding NavigateCommand}"
                CommandParameter="{x:Type local:ImageButtonDemoPage}" />

        </FlexLayout>
    </ScrollView>
</ContentPage>

```

Each `Button` has its `Command` property bound to a property named `NavigateCommand`, and the `CommandParameter` is set to a `Type` object corresponding to one of the page classes in the project.

That `NavigateCommand` property is of type `ICommand` and is defined in the code-behind file:

```

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(async (Type pageType) =>
    {
        Page page = (Page)Activator.CreateInstance(pageType);
        await Navigation.PushAsync(page);
    });
}

    BindingContext = this;
}

public ICommand NavigateCommand { private set; get; }
}

```

The constructor initializes the `NavigateCommand` property to a `Command<Type>` object because `Type` is the type of the `CommandParameter` object set in the XAML file. This means that the `execute` method has an argument of type `Type` that corresponds to this `CommandParameter` object. The function instantiates the page and then navigates to it.

Notice that the constructor concludes by setting its `BindingContext` to itself. This is necessary for properties in the XAML file to bind to the `NavigateCommand` property.

## Pressing and releasing the button

Besides the `Clicked` event, `Button` also defines `Pressed` and `Released` events. The `Pressed` event occurs when a finger presses on a `Button`, or a mouse button is pressed with the pointer positioned over the `Button`. The `Released` event occurs when the finger or mouse button is released. Generally, a `Clicked` event is also fired at the same time as the `Released` event, but if the finger or mouse pointer slides away from the surface of the `Button` before being released, the `Clicked` event might not occur.

The `Pressed` and `Released` events are not often used, but they can be used for special purposes, as demonstrated in the **Press and Release Button** page. The XAML file contains a `Label` and a `Button` with handlers attached for the `Pressed` and `Released` events:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ButtonDemos.PressAndReleaseButtonPage"
    Title="Press and Release Button">
<StackLayout>

    <Label x:Name="label"
        Text="Press and hold the Button below"
        FontSize="Large"
        VerticalOptions="CenterAndExpand"
        HorizontalOptions="Center" />

    <Button Text="Press to Rotate Text!"
        VerticalOptions="CenterAndExpand"
        HorizontalOptions="Center"
        Pressed="OnButtonPressed"
        Released="OnButtonReleased" />

</StackLayout>
</ContentPage>
```

The code-behind file animates the `Label` when a `Pressed` event occurs, but suspends the rotation when a `Released` event occurs:

```

public partial class PressAndReleaseButtonPage : ContentPage
{
    bool animationInProgress = false;
    Stopwatch stopwatch = new Stopwatch();

    public PressAndReleaseButtonPage ()
    {
        InitializeComponent ();
    }

    void OnButtonPressed(object sender, EventArgs args)
    {
        stopwatch.Start();
        animationInProgress = true;

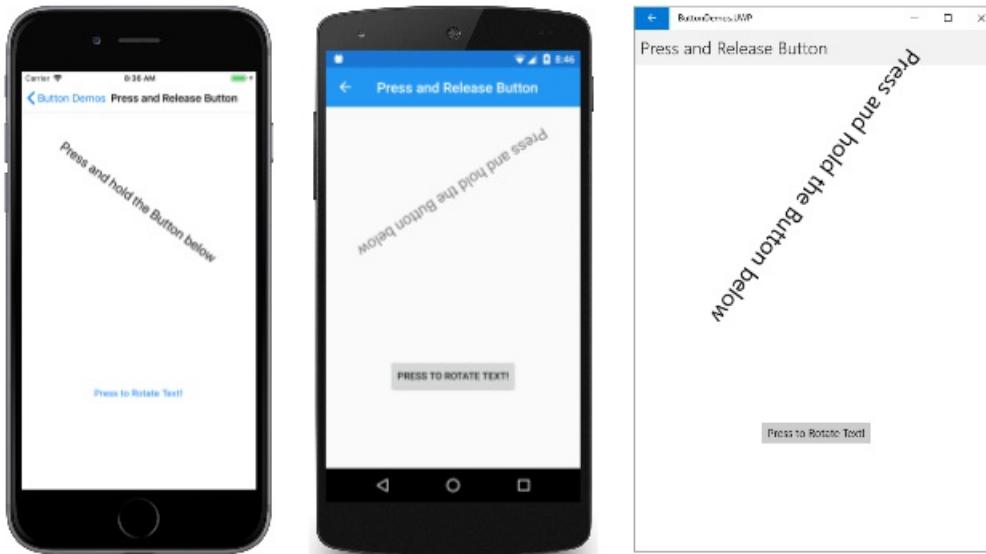
        Device.StartTimer(TimeSpan.FromMilliseconds(16), () =>
        {
            label.Rotation = 360 * (stopwatch.Elapsed.TotalSeconds % 1);

            return animationInProgress;
        });
    }

    void OnButtonReleased(object sender, EventArgs args)
    {
        animationInProgress = false;
        stopwatch.Stop();
    }
}

```

The result is that the `Label` only rotates while a finger is in contact with the `Button`, and stops when the finger is released:



This kind of behavior has applications for games: A finger held on a `Button` might make an on screen object move in a particular direction.

## Button appearance

The `Button` inherits or defines several properties that affect its appearance:

- `TextColor` is the color of the `Button` text
- `BackgroundColor` is the color of the background to that text
- `BorderColor` is the color of an area surrounding the `Button`

- `FontFamily` is the font family used for the text
- `FontSize` is the size of the text
- `FontAttributes` indicates if the text is italic or bold
- `BorderWidth` is the width of the border
- `CornerRadius` is the corner radius of the `Button`

#### NOTE

The `Button` class also has `Margin` and `Padding` properties that control the layout behavior of the `Button`. For more information, see [Margin and Padding](#).

The effects of six of these properties (excluding `FontFamily` and `FontAttributes`) are demonstrated in the **Button Appearance** page. Another property, `Image`, is discussed in the section [Using bitmaps with button](#).

All of the views and data bindings in the **Button Appearance** page are defined in the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ButtonDemos"
    x:Class="ButtonDemos.ButtonAppearancePage"
    Title="Button Appearance">

    <StackLayout>
        <Button x:Name="button"
            Text="Button"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center"
            TextColor="{Binding Source={x:Reference textColorPicker},
                Path=SelectedItem.Color}"
            BackgroundColor="{Binding Source={x:Reference backgroundColorPicker},
                Path=SelectedItem.Color}"
            BorderColor="{Binding Source={x:Reference borderColorPicker},
                Path=SelectedItem.Color}" />

        <StackLayout BindingContext="{x:Reference button}"
            Padding="10">

            <Slider x:Name="fontSizeSlider"
                Maximum="48"
                Minimum="1"
                Value="{Binding FontSize}" />

            <Label Text="{Binding Source={x:Reference fontSizeSlider},
                Path=Value,
                StringFormat='FontSize = {0:F0}'}"
                HorizontalTextAlignment="Center" />

            <Slider x:Name="borderWidthSlider"
                Minimum="-1"
                Maximum="12"
                Value="{Binding BorderWidth}" />

            <Label Text="{Binding Source={x:Reference borderWidthSlider},
                Path=Value,
                StringFormat='BorderWidth = {0:F0}'}"
                HorizontalTextAlignment="Center" />

            <Slider x:Name="cornerRadiusSlider"
                Minimum="-1"
                Maximum="24"
                Value="{Binding CornerRadius}" />

            <Label Text="{Binding Source={x:Reference cornerRadiusSlider},
                Path=Value".
```

```

        Value,
        StringFormat='CornerRadius = {0:F0}'}"
        HorizontalTextAlignment="Center" />

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Grid.Resources>
        <Style TargetType="Label">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
    </Grid.Resources>

    <Label Text="Text Color:">
        Grid.Row="0" Grid.Column="0" />

    <Picker x:Name="textColorPicker"
        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
        ItemDisplayBinding="{Binding FriendlyName}"
        SelectedIndex="0"
        Grid.Row="0" Grid.Column="1" />

    <Label Text="Background Color:">
        Grid.Row="1" Grid.Column="0" />

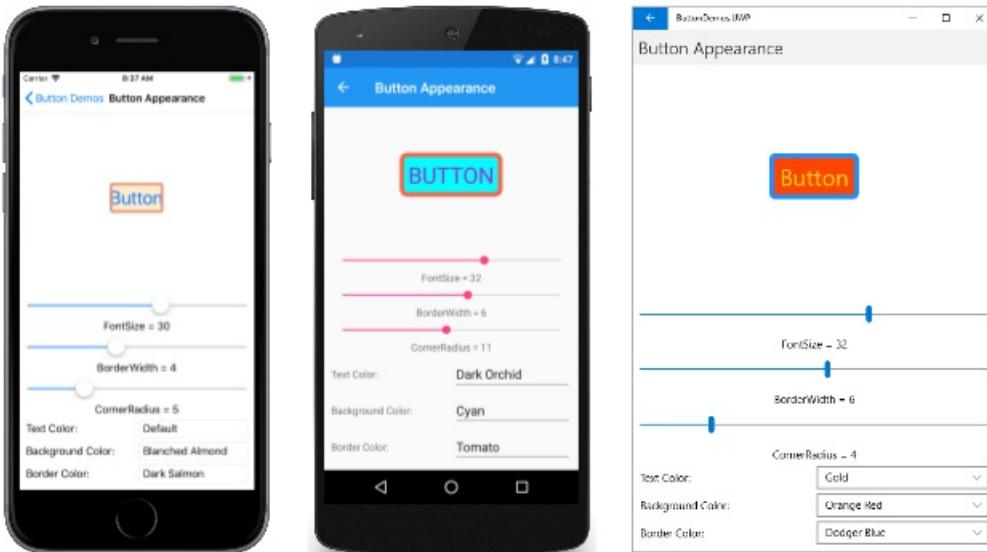
    <Picker x:Name="backgroundColorPicker"
        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
        ItemDisplayBinding="{Binding FriendlyName}"
        SelectedIndex="0"
        Grid.Row="1" Grid.Column="1" />

    <Label Text="Border Color:">
        Grid.Row="2" Grid.Column="0" />

    <Picker x:Name="borderColorPicker"
        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
        ItemDisplayBinding="{Binding FriendlyName}"
        SelectedIndex="0"
        Grid.Row="2" Grid.Column="1" />
</Grid>
</StackLayout>
</StackLayout>
</ContentPage>
```

The `Button` at the top of the page has its three `Color` properties bound to `Picker` elements at the bottom of the page. The items in the `Picker` elements are colors from the `NamedColor` class included in the project. Three `Slider` elements contain two-way bindings to the `FontSize`, `BorderWidth`, and `CornerRadius` properties of the `Button`.

This program allows you to experiment with combinations of all these properties:



To see the `Button` border, you'll need to set a `BorderColor` to something other than `Default`, and the `BorderWidth` to a positive value.

On iOS, you'll notice that large border widths intrude into the interior of the `Button` and interfere with the display of text. If you choose to use a border with an iOS `Button`, you'll probably want to begin and end the `Text` property with spaces to retain its visibility.

On UWP, selecting a `CornerRadius` that exceeds half the height of the `Button` raises an exception.

## Button visual states

`Button` has a `Pressed` `VisualState` that can be used to initiate a visual change to the `Button` when pressed by the user, provided that it's enabled.

The following XAML example shows how to define a visual state for the `Pressed` state:

```
<Button Text="Click me!">
    ...
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="Scale"
                           Value="1" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Pressed">
                <VisualState.Setters>
                    <Setter Property="Scale"
                           Value="0.8" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</ImageButton>
```

The `Pressed` `VisualState` specifies that when the `Button` is pressed, its `Scale` property will be changed from its default value of 1 to 0.8. The `Normal` `VisualState` specifies that when the `Button` is in a normal state, its `Scale` property will be set to 1. Therefore, the overall effect is that when the `Button` is pressed, it's rescaled to be slightly smaller, and when the `Button` is released, it's rescaled to its default size.

For more information about visual states, see [The Xamarin.Forms Visual State Manager](#).

## Creating a toggle button

It is possible to subclass `Button` so that it works like an on-off switch: Tap the button once to toggle the button on and tap it again to toggle it off.

The following `ToggleButton` class derives from `Button` and defines a new event named `Toggled` and a Boolean property named `IsToggled`. These are the same two properties defined by the `Xamarin.Forms Switch`:

```
class ToggleButton : Button
{
    public event EventHandler<ToggledEventArgs> Toggled;

    public static BindableProperty IsToggledProperty =
        BindableProperty.Create("IsToggled", typeof(bool), typeof(ToggleButton), false,
                               propertyChanged: OnIsToggledChanged);

    public ToggleButton()
    {
        Clicked += (sender, args) => IsToggled ^= true;
    }

    public bool IsToggled
    {
        set { SetValue(IsToggledProperty, value); }
        get { return (bool)GetValue(IsToggledProperty); }
    }

    protected override void OnParentSet()
    {
        base.OnParentSet();
        VisualStateManager.GoToState(this, "ToggledOff");
    }

    static void OnIsToggledChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ToggleButton toggleButton = (ToggleButton)bindable;
        bool isToggled = (bool)newValue;

        // Fire event
        toggleButton.Toggled?.Invoke(toggleButton, new ToggledEventArgs(isToggled));

        // Set the visual state
        VisualStateManager.GoToState(toggleButton, isToggled ? "ToggledOn" : "ToggledOff");
    }
}
```

The `ToggleButton` constructor attaches a handler to the `Clicked` event so that it can change the value of the `IsToggled` property. The `OnIsToggledChanged` method fires the `Toggled` event.

The last line of the `OnIsToggledChanged` method calls the static `VisualStateManager.GoToState` method with the two text strings "ToggledOn" and "ToggledOff". You can read about this method and how your application can respond to visual states in the article [The Xamarin.Forms Visual State Manager](#).

Because `ToggleButton` makes the call to `VisualStateManager.GoToState`, the class itself doesn't need to include any additional facilities to change the button's appearance based on its `IsToggled` state. That is the responsibility of the XAML that hosts the `ToggleButton`.

The **Toggle Button Demo** page contains two instances of `ToggleButton`, including Visual State Manager markup that sets the `Text`, `BackgroundColor`, and `TextColor` of the button based on the visual state:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ButtonDemos"
    x:Class="ButtonDemos.ToggleButtonDemoPage"
    Title="Toggle Button Demo">

    <ContentPage.Resources>
        <Style TargetType="local:ToggleButton">
            <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            <Setter Property="HorizontalOptions" Value="Center" />
        </Style>
    </ContentPage.Resources>

    <StackLayout Padding="10, 0">
        <local:ToggleButton Toggled="OnItalicButtonToggled">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ToggleStates">
                    <VisualState Name="ToggledOff">
                        <VisualState.Setters>
                            <Setter Property="Text" Value="Italic Off" />
                            <Setter Property="BackgroundColor" Value="#C0C0C0" />
                            <Setter Property="TextColor" Value="Black" />
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState Name="ToggledOn">
                        <VisualState.Setters>
                            <Setter Property="Text" Value=" Italic On " />
                            <Setter Property="BackgroundColor" Value="#404040" />
                            <Setter Property="TextColor" Value="White" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </local:ToggleButton>

        <local:ToggleButton Toggled="OnBoldButtonToggled">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ToggleStates">
                    <VisualState Name="ToggledOff">
                        <VisualState.Setters>
                            <Setter Property="Text" Value="Bold Off" />
                            <Setter Property="BackgroundColor" Value="#C0C0C0" />
                            <Setter Property="TextColor" Value="Black" />
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState Name="ToggledOn">
                        <VisualState.Setters>
                            <Setter Property="Text" Value=" Bold On " />
                            <Setter Property="BackgroundColor" Value="#404040" />
                            <Setter Property="TextColor" Value="White" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </local:ToggleButton>

        <Label x:Name="label"
            Text="Just a little passage of some sample text that can be formatted in italic or boldface by
            toggling the two buttons."
            FontSize="Large"
            HorizontalTextAlignment="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

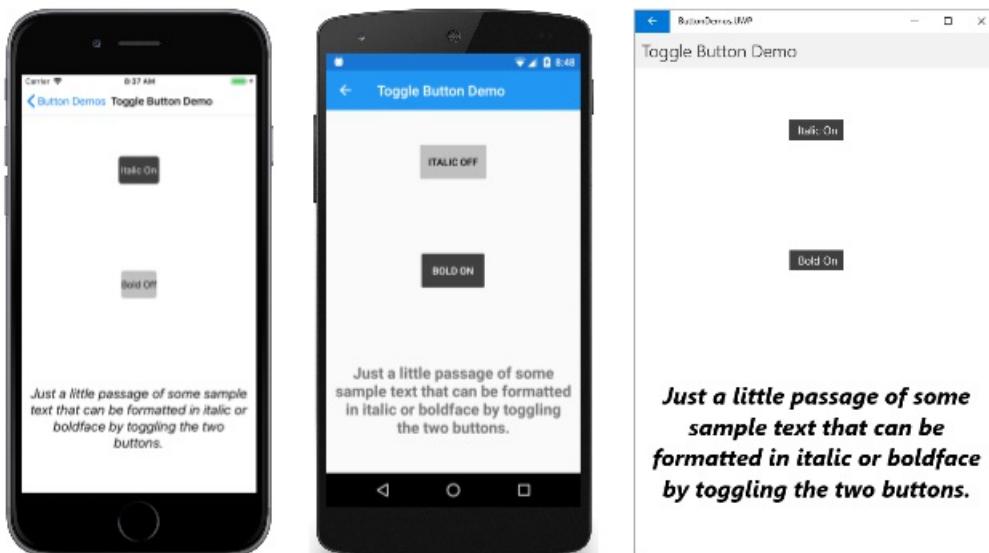
The `Toggled` event handlers are in the code-behind file. They are responsible for setting the `FontAttributes` property of the `Label` based on the state of the buttons:

```
public partial class ToggleButtonDemoPage : ContentPage
{
    public ToggleButtonDemoPage ()
    {
        InitializeComponent ();
    }

    void OnItalicButtonToggled(object sender, ToggledEventArgs args)
    {
        if (args.Value)
        {
            label.FontAttributes |= FontAttributes.Italic;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Italic;
        }
    }

    void OnBoldButtonToggled(object sender, ToggledEventArgs args)
    {
        if (args.Value)
        {
            label.FontAttributes |= FontAttributes.Bold;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Bold;
        }
    }
}
```

Here's the program running on iOS, Android, and the UWP:



## Using bitmaps with buttons

The `Button` class defines an `Image` property that allows you to display a bitmap image on the `Button`, either alone or in combination with text. You can also specify how the text and image are arranged.

The `Image` property is of type `FileImageSource`, which means that the bitmaps must be stored as resources in the

individual platform projects, and not in the .NET Standard library project.

Each platform supported by Xamarin.Forms allows images to be stored in multiple sizes for different pixel resolutions of the various devices that the application might run on. These multiple bitmaps are named or stored in such a way that the operating system can pick the best match for the device's video display resolution.

For a bitmap on a `Button`, the best size is usually between 32 and 64 device-independent units, depending on how large you want it to be. The images used in this example are based on a size of 48 device-independent units.

In the iOS project, the **Resources** folder contains three sizes of this image:

- A 48-pixel square bitmap stored as `/Resources/MonkeyFace.png`
- A 96-pixel square bitmap stored as `/Resource/MonkeyFace@2x.png`
- A 144-pixel square bitmap stored as `/Resource/MonkeyFace@3x.png`

All three bitmaps were given a **Build Action** of **BundleResource**.

For the Android project, the bitmaps all have the same name, but they are stored in different subfolders of the **Resources** folder:

- A 72-pixel square bitmap stored as `/Resources/drawable-hdpi/MonkeyFace.png`
- A 96-pixel square bitmap stored as `/Resources/drawable-xhdpi/MonkeyFace.png`
- A 144-pixel square bitmap stored as `/Resources/drawable-xxhdpi/MonkeyFace.png`
- A 192-pixel square bitmap stored as `/Resources/drawable-xxxhdpi/MonkeyFace.png`

These were given a **Build Action** of **AndroidResource**.

In the UWP project, bitmaps can be stored anywhere in the project, but they are generally stored in a custom folder or the **Assets** existing folder. The UWP project contains these bitmaps:

- A 48-pixel square bitmap stored as `/Assets/MonkeyFace.scale-100.png`
- A 96-pixel square bitmap stored as `/Assets/MonkeyFace.scale-200.png`
- A 192-pixel square bitmap stored as `/Assets/MonkeyFace.scale-400.png`

They were all given a **Build Action** of **Content**.

You can specify how the `Text` and `Image` properties are arranged on the `Button` using the `ContentLayout` property of `Button`. This property is of type `ButtonContentLayout`, which is an embedded class in `Button`. The `constructor` has two arguments:

- A member of the `ImagePosition` enumeration: `Left`, `Top`, `Right`, or `Bottom` indicating how the bitmap appears relative to the text.
- A `double` value for the spacing between the bitmap and the text.

The defaults are `Left` and 10 units. Two read-only properties of `ButtonContentLayout` named `Position` and `Spacing` provide the values of those properties.

In code, you can create a `Button` and set the `ContentLayout` property like this:

```
Button button = new Button
{
    Text = "button text",
    Image = new FileImageSource
    {
        File = "image filename"
    },
    ContentLayout = new Button.ButtonContentLayout(Button.ButtonContentLayout.ImagePosition.Right, 20)
};
```

In XAML, you need specify only the enumeration member, or the spacing, or both in any order separated by commas:

```
<Button Text="button text"  
       Image="image filename"  
       ContentLayout="Right, 20" />
```

The **Image Button Demo** page uses `OnPlatform` to specify different filenames for the iOS, Android, and UWP bitmap files. If you want to use the same filename for each platform and avoid the use of `OnPlatform`, you'll need to store the UWP bitmaps in the root directory of the project.

The first `Button` on the **Image Button Demo** page sets the `Image` property but not the `Text` property:

```
<Button>  
  <Button.Image>  
    <OnPlatform x:TypeArguments="FileImageSource">  
      <On Platform="iOS, Android" Value="MonkeyFace.png" />  
      <On Platform="UWP" Value="Assets/MonkeyFace.png" />  
    </OnPlatform>  
  </Button.Image>  
</Button>
```

If the UWP bitmaps are stored in the root directory of the project, this markup can be considerably simplified:

```
<Button Image="MonkeyFace.png" />
```

To avoid a lot of repetitious markup in the **ImageButtonDemo.xaml** file, an implicit `style` is also defined to set the `Image` property. This `style` is automatically applied to five other `Button` elements. Here's the complete XAML file:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ButtonDemos.ImageButtonDemoPage">

    <FlexLayout Direction="Column"
        JustifyContent="SpaceEvenly"
        AlignItems="Center">

        <FlexLayout.Resources>
            <Style TargetType="Button">
                <Setter Property="Image">
                    <OnPlatform x:TypeArguments="FileImageSource">
                        <On Platform="iOS, Android" Value="MonkeyFace.png" />
                        <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                    </OnPlatform>
                </Setter>
            </Style>
        </FlexLayout.Resources>

        <Button>
            <Button.Image>
                <OnPlatform x:TypeArguments="FileImageSource">
                    <On Platform="iOS, Android" Value="MonkeyFace.png" />
                    <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                </OnPlatform>
            </Button.Image>
        </Button>

        <Button Text="Default" />

        <Button Text="Left - 10"
            ContentLayout="Left, 10" />

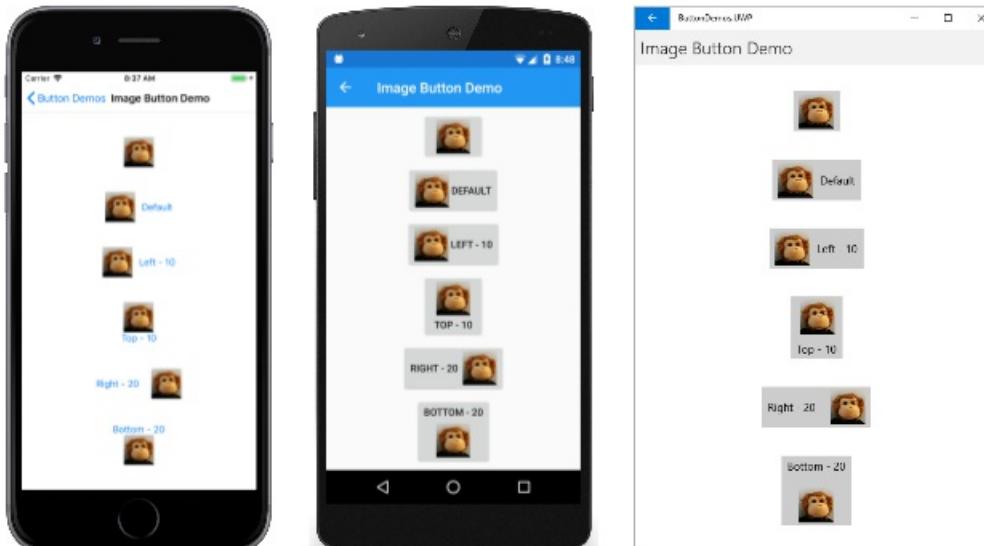
        <Button Text="Top - 10"
            ContentLayout="Top, 10" />

        <Button Text="Right - 20"
            ContentLayout="Right, 20" />

        <Button Text="Bottom - 20"
            ContentLayout="Bottom, 20" />
    </FlexLayout>
</ContentPage>

```

The final four `Button` elements make use of the `ContentLayout` property to specify a position and spacing of the text and bitmap:



You've now seen the various ways that you can handle `Button` events and change the `Button` appearance.

## Related links

- [ButtonDemos sample](#)
- [Button API](#)

# Colors in Xamarin.Forms

10/11/2018 • 3 minutes to read • [Edit Online](#)

Xamarin.Forms provides a flexible cross-platform `Color` class.

This article introduces the various ways the `Color` class can be used in Xamarin.Forms.

The `Color` class provides a number of methods to build a color instance

- **Named Colors** - a collection of common named-colors, including `Red`, `Green`, and `Blue`.
- **FromHex** - string value similar to the syntax used in HTML, eg "00FF00". Alpha is can optionally be specified as the first pair of characters ("CC00FF00").
- **FromHsla** - Hue, saturation and luminosity `double` values, with optional alpha value (0.0-1.0).
- **FromRgb** - Red, green, and blue `int` values (0-255).
- **FromRgba** - Red, green, blue, and alpha `int` values (0-255).
- **FromUint** - set a single `double` value representing `argb`.

Here's some example colors, assigned to the `BackgroundColor` of some labels using different variations of the allowed syntax:

```
var red    = new Label { Text = "Red",   BackgroundColor = Color.Red };
var orange = new Label { Text = "Orange",BackgroundColor = Color.FromHex("FF6A00") };
var yellow = new Label { Text = "Yellow",BackgroundColor = Color.FromHsla(0.167, 1.0, 0.5, 1.0) };
var green  = new Label { Text = "Green",  BackgroundColor = Color.FromRgb (38, 127, 0) };
var blue   = new Label { Text = "Blue",   BackgroundColor = Color.FromRgba(0, 38, 255, 255) };
var indigo = new Label { Text = "Indigo",BackgroundColor = Color.FromRgb (0, 72, 255) };
var violet = new Label { Text = "Violet",BackgroundColor = Color.FromHsla(0.82, 1, 0.25, 1) };

var transparent = new Label { Text = "Transparent",BackgroundColor = Color.Transparent };
var @default = new Label { Text = "Default",   BackgroundColor = Color.Default };
var accent = new Label { Text = "Accent",     BackgroundColor = Color.Accent };
```

These colors are shown on each platform below. Notice the final color - `Accent` - is a blue-ish color for iOS and Android; this value is defined by Xamarin.Forms.



## Color.Default

Use the `Default` to set (or re-set) a color value back to the platform default (understanding that this represents a

different underlying color on each platform for each property).

Developers can use this value to set a `Color` property but should **not** query this instance for its component RGB values (they're all set to -1).

## Color.Transparent

Set the color to clear.

## Color.Accent

On iOS and Android this instance is set to a contrasting color that is visible on the default background but is not the same as the default text color.

## Additional Methods

`Color` instances include additional methods that can be used to create new colors:

- **AddLuminosity** - returns a new color by modifying the luminosity by the supplied delta.
- **WithHue** - returns a new color, replacing the hue with the value supplied.
- **WithLuminosity** - returns a new color, replacing the luminosity with the value supplied.
- **WithSaturation** - returns a new color, replacing the saturation with the value supplied.
- **MultiplyAlpha** - returns a new color by modifying the alpha, multiplying it by the supplied alpha value.

## Implicit Conversions

Implicit conversion between the `Xamarin.Forms.Color` and `System.Drawing.Color` types can be performed:

```
Xamarin.Forms.Color xfColor = Xamarin.Forms.Color.FromRgb(0, 72, 255);
System.Drawing.Color sdColor = System.Drawing.Color.FromArgb(38, 127, 0);

// Implicitly convert from a Xamarin.Forms.Color to a System.Drawing.Color
System.Drawing.Color sdColor2 = xfColor;

// Implicitly convert from a System.Drawing.Color to a Xamarin.Forms.Color
Xamarin.Forms.Color xfColor2 = sdColor;
```

## Device.RuntimePlatform

This code snippet uses the `Device.RuntimePlatform` property to selectively set the color of an `ActivityIndicator`:

```
ActivityIndicator activityIndicator = new ActivityIndicator
{
    Color = Device.RuntimePlatform == Device.iOS ? Color.Black : Color.Default,
    IsRunning = true
};
```

## Using from XAML

Colors can also be easily referenced in XAML using the defined color names or the Hex representations shown here:

```
<Label Text="Sea color" BackgroundColor="Aqua" />
<Label Text="RGB" BackgroundColor="#00FF00" />
<Label Text="Alpha plus RGB" BackgroundColor="#CC00FF00" />
<Label Text="Tiny RGB" BackgroundColor="#0F0" />
<Label Text="Tiny Alpha plus RGB" BackgroundColor="#C0F0" />
```

#### NOTE

When using XAML compilation, color names are case insensitive and therefore can be written in lowercase. For more information about XAML compilation, see [XAML Compilation](#).

## Summary

The Xamarin.Forms `Color` class is used to create platform-aware color references. It can be used in shared code and XAML.

## Related Links

- [ColorsSample](#)
- [Bindable Picker \(sample\)](#)

# Controls Reference

7/12/2018 • 2 minutes to read • [Edit Online](#)

*A description of all the visual elements used to construct a Xamarin.Forms application.*

The visual interface of a Xamarin.Forms application is constructed of objects that map to the native controls of each target platform. This allows platform-specific applications for iOS, Android, and the Universal Windows Platform to use Xamarin.Forms code contained in a [.NET Standard library](#) or a [Shared Project](#).

The four main control groups used to create the user interface of a Xamarin.Forms application are shown in these four articles:

- [Pages](#)
- [Layouts](#)
- [Views](#)
- [Cells](#)

A Xamarin.Forms page generally occupies the entire screen. The page usually contains a layout, which contains views and possibly other layouts. Cells are specialized components used in connection with [TableView](#) and [ListView](#).

In the four articles on [Pages](#), [Layouts](#), [Views](#), and [Cells](#), each type of control is described with links to its API documentation, an article describing its use (if one exists), and one or more sample programs (if they exist). Each type of control is also accompanied by a screenshot showing a page from the [FormsGallery](#) sample running on iOS, Android, and UWP devices. Below each screenshot are links to the source code for the C# page, the equivalent XAML page, and (when appropriate) the C# code-behind file for the XAML page.

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms FormsGallery sample](#)
- [API Documentation](#)

# Xamarin.Forms Pages

7/12/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms Pages represent cross-platform mobile application screens.*

All the page types that are described below derive from the Xamarin.Forms [Page](#) class. These visual elements occupy all or most of the screen. A [Page](#) object represents a [ViewController](#) in iOS and a [Page](#) in the Universal Windows Platform. On Android, each page takes up the screen like an [Activity](#), but Xamarin.Forms pages are *not* [Activity](#) objects.



ContentPage    MasterDetailPage    NavigationPage    TabbedPage    TemplatedPage    CarouselPage

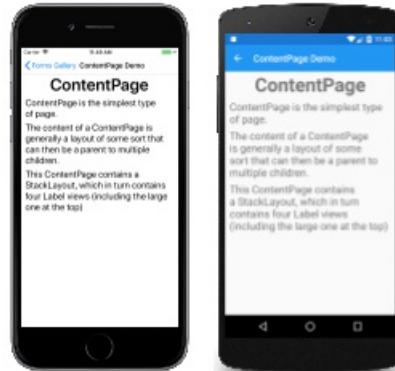
## Pages

Xamarin.Forms supports the following page types:

### ContentPage

[ContentPage](#) is the simplest and most common type of page. Set the [Content](#) property to a single [View](#) object, which is most often a [Layout](#) such as [StackLayout](#), [Grid](#), or [ScrollView](#).

[API Documentation](#)

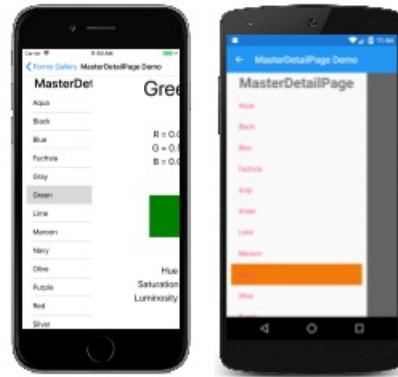


[C# code for this page / XAML page](#)

### MasterDetailPage

A `MasterDetailPage` manages two panes of information. Set the `Master` property to a page generally showing a list or menu. Set the `Detail` property to a page showing a selected item from the master page. The `IsPresented` property governs whether the master or detail page is visible.

[API Documentation](#) / [Guide](#) / [Sample](#)

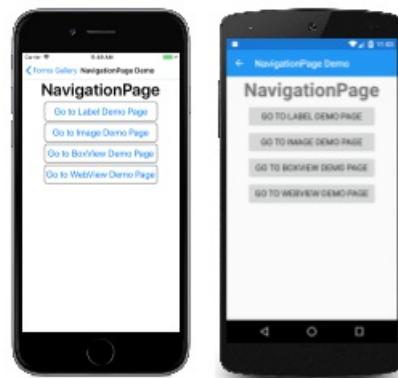


[C# code for this page](#) / [XAML page](#) with code-behind

## NavigationPage

The `NavigationPage` manages navigation among other pages using a stack-based architecture. When using page navigation in your application, an instance of the home page should be passed to the constructor of a `NavigationPage` object.

[API Documentation](#) / [Guide](#) / [Sample 1, 2, and 3](#)



[C# code for this page](#) / [XAML Page](#) with code=behind

## TabbedPage

`TabbedPage` derives from the abstract `MultiPage` class and allows navigation among child pages using tabs. Set the `Children` property to a collection of pages, or set the `ItemsSource` property to a collection of data objects and the `ItemTemplate` property to a `DataTemplate` describing how each object is to be visually represented.

[API Documentation](#) / [Guide](#) / [Sample 1 and 2](#)



[C# code for this page](#) / [XAML page](#)

## CarouselPage

`CarouselPage` derives from the abstract `MultiPage` class and allows navigation among child pages through finger swiping. Set the `Children` property to a collection of `ContentPage` objects, or set the `ItemsSource` property to a collection of data objects and the `ItemTemplate` property to a `DataTemplate` describing how each object is to be visually represented.

[API Documentation](#) / [Guide](#) / [Sample 1](#) and [2](#)

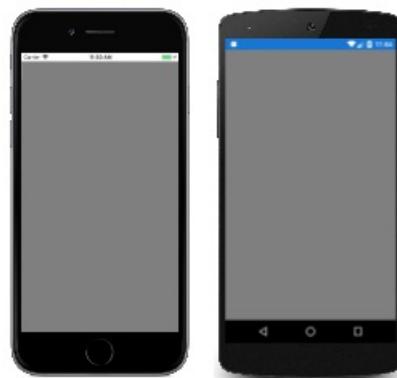


[C# code for this page](#) / [XAML page](#)

## TempledPage

`TempledPage` displays full-screen content with a control template, and is the base class for `ContentPage`.

[API Documentation](#) / [Guide](#)



## Related Links

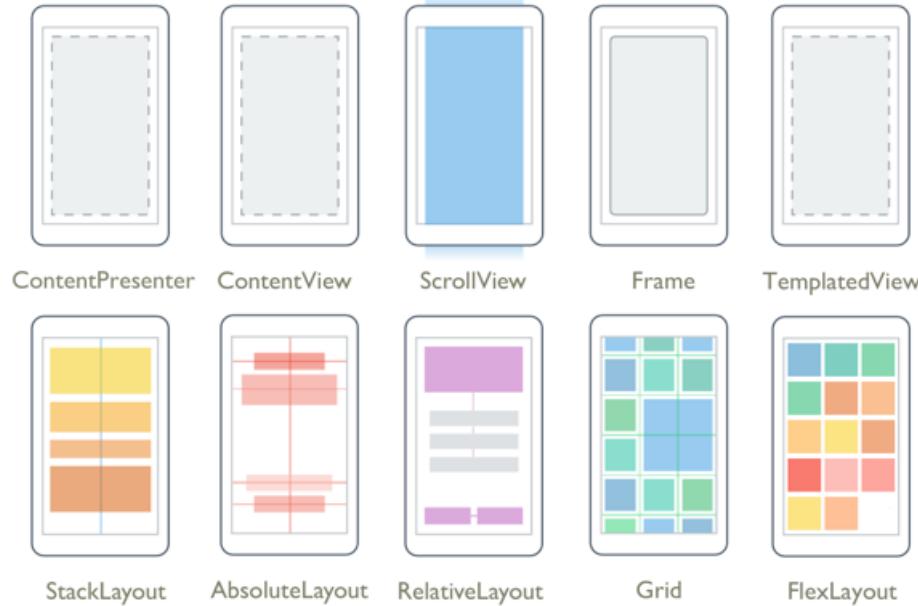
- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms Layouts

7/12/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms Layouts are used to compose user-interface controls into visual structures.*

The [Layout](#) and [Layout<T>](#) classes in Xamarin.Forms are specialized subtypes of views that act as containers for views and other layouts. The [Layout](#) class itself derives from [View](#). A [Layout](#) derivative typically contains logic to set the position and size of child elements in Xamarin.Forms applications.



The classes that derive from [Layout](#) can be divided into two categories:

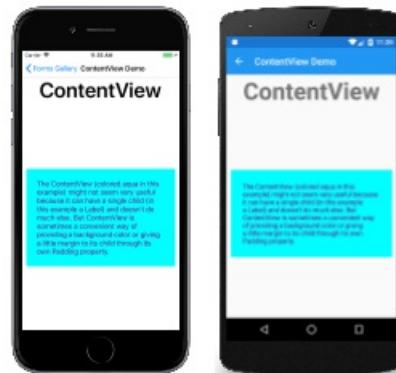
## Layouts with Single Content

These classes derive from [Layout](#), which defines [Padding](#) and [IsClippedToBounds](#) properties.

### ContentView

[ContentView](#) contains a single child that is set with the [Content](#) property. The [content](#) property can be set to any [View](#) derivative, including other [Layout](#) derivatives. [ContentView](#) is mostly used as a structural element and serves as a base class to [Frame](#).

[API Documentation](#)

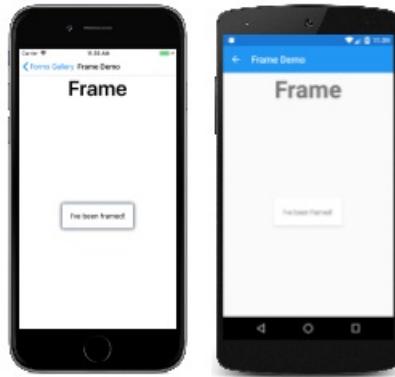


[C# code for this page / XAML page](#)

### Frame

The `Frame` class derives from `ContentView` and displays a rectangular frame around its child. `Frame` has a default `Padding` value of 20, and also defines `OutlineColor`, `CornerRadius`, and `HasShadow` properties.

[API Documentation](#)



[C# code for this page / XAML page](#)

## ScrollView

`ScrollView` is capable of scrolling its contents. Set the `Content` property to a view or layout too large to fit on the screen. (The content of a `ScrollView` is very often a `StackLayout`.) Set the `Orientation` property to indicate if scrolling should be vertical, horizontal, or both.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

## TempledView

`TempledView` displays content with a control template, and is the base class for `ContentView`.

[API Documentation / Guide](#)



## ContentPresenter

**ContentPresenter** is a layout manager for templated views, used within a **ControlTemplate** to mark where the content that is to be presented appears.

[API Documentation / Guide](#)



## Layouts with Multiple Children

These classes derive from **Layout<View>**.

### StackLayout

**StackLayout** positions child elements in a stack either horizontally or vertically based on the **Orientation** property. The **Spacing** property governs the spacing between the children, and has a default value of 6.

[API Documentation / Guide / Sample](#)



[C# code for this page](#) / [XAML page](#)

### Grid

**Grid** positions its child elements in a grid of rows and columns. A child's position is indicated using the attached properties **Row**, **Column**, **RowSpan**, and **ColumnSpan**.

[API Documentation / Guide / Sample](#)



[C# code for this page](#) / [XAML page](#)

### AbsoluteLayout

`AbsoluteLayout` positions child elements at specific locations relative to its parent. A child's position is indicated using the attached properties `LayoutBounds` and `LayoutFlags`. An `AbsoluteLayout` is useful for animating the positions of views.

[API Documentation](#) / [Guide](#) / [Sample](#)

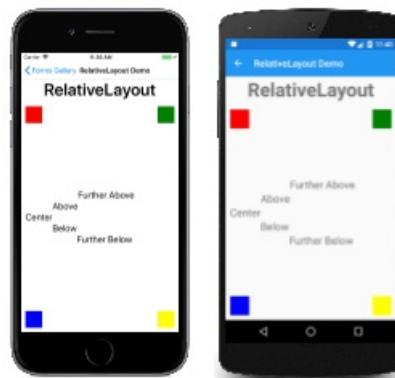


[C# code for this page](#) / [XAML page](#) with code-behind

## RelativeLayout

`RelativeLayout` positions child elements relative to the `RelativeLayout` itself or to their siblings. A child's position is indicated using the attached properties that are set to objects of type `Constraint` and `BoundsConstraint`.

[API Documentation](#) / [Guide](#) / [Sample](#)

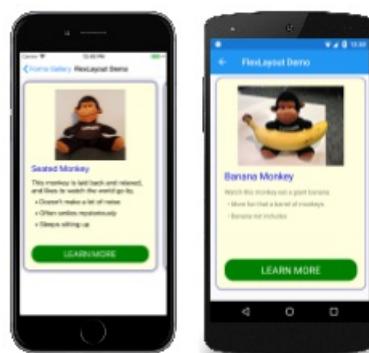


[C# code for this page](#) / [XAML page](#)

## FlexLayout

`FlexLayout` is based on the CSS [Flexible Box Layout Module](#), commonly known as *flex layout* or *flex-box*. `FlexLayout` defines six bindable properties and five attached bindable properties that allow children to be stacked or wrapped with many alignment and orientation options.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#)

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)

- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms Views

11/20/2018 • 6 minutes to read • [Edit Online](#)

*Xamarin.Forms views are the building blocks of cross-platform mobile user interfaces.*

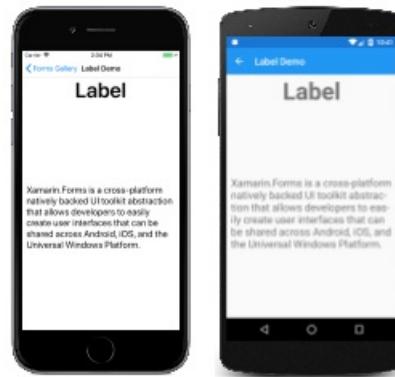
Views are user-interface objects such as labels, buttons, and sliders that are commonly known as *controls* or *widgets* in other graphical programming environments. The views supported by Xamarin.Forms all derive from the [View](#) class. They can be divided into several categories:

## Views for presentation

### Label

[Label](#) displays single-line text strings or multi-line blocks of text, either with constant or variable formatting. Set the [Text](#) property to a string for constant formatting, or set the [FormattedText](#) property to a [FormattedString](#) object for variable formatting.

[API Documentation](#) / [Guide](#) / [Sample](#)

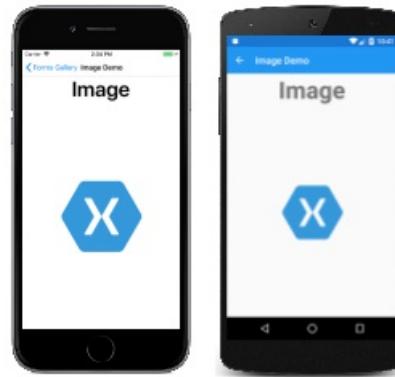


[C# code for this page](#) / [XAML page](#)

### Image

[Image](#) displays a bitmap. Bitmaps can be downloaded over the Web, embedded as resources in the common project or platform projects, or created using a .NET [Stream](#) object.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#)

### BoxView

`BoxView` displays a solid rectangle colored by the `Color` property. `BoxView` has a default size request of 40x40. For other sizes, assign the `WidthRequest` and `HeightRequest` properties.

[API Documentation](#) / [Guide](#) / [Sample 1, 2, 3, 4, 5, and 6](#)



[C# code for this page](#) / [XAML page](#)

## WebView

`WebView` displays Web pages or HTML content, based on whether the `Source` property is set to a `UriWebViewSource` or an `HtmlWebViewSource` object.

[API Documentation](#) / [Guide](#) / [Sample 1 and 2](#)



[C# code for this page](#) / [XAML page](#)

## OpenGLView

`OpenGLView` displays OpenGL graphics in iOS and Android projects. There is no support for the Universal Windows Platform. The iOS and Android projects require a reference to the **OpenTK-1.0** assembly or the **OpenTK** version 1.0.0.0 assembly. `OpenGLView` is easier to use in a Shared Project; if used in a .NET Standard library, then a Dependency Service will also be required (as shown in the sample code).

This is the only graphics facility that is built into Xamarin.Forms, but a Xamarin.Forms application can also render graphics using `CocosSharp`, `SkiaSharp`, or `UrhoSharp`.

[API Documentation](#)

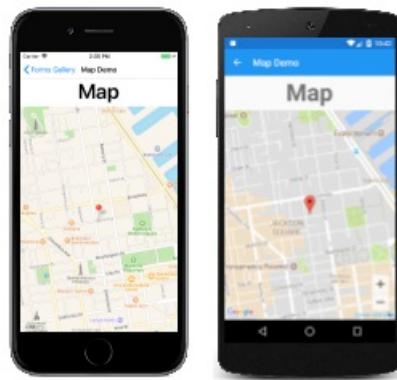


[C# code for this page](#) / [XAML page with code-behind](#)

## Map

**Map** displays a map. The **Xamarin.Forms.Maps** Nuget package must be installed. Android and Universal Windows Platform require a map authorization key.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#)

## Views that initiate commands

### Button

**Button** is a rectangular object that displays text, and which fires a **Clicked** event when it's been pressed.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#) with code-behind

### ImageButton

**ImageButton** is a rectangular object that displays an image, and which fires a **Clicked** event when it's been pressed.

[Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#) with code-behind

### SearchBar

`SearchBar` displays an area for the user to type a text string, and a button (or a keyboard key) that signals the application to perform a search. The `Text` property provides access to the text, and the `SearchButtonPressed` event indicates that the button has been pressed.

[API Documentation](#)



[C# code for this page / XAML page with code-behind](#)

## Views for setting values

### Slider

`Slider` allows the user to select a `double` value from a continuous range specified with the `Minimum` and `Maximum` properties.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

### Stepper

`Stepper` allows the user to select a `double` value from a range of incremental values specified with the `Minimum`, `Maximum`, and `Increment` properties.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

### Switch

**Switch** takes the form of an on/off switch to allow the user to select a Boolean value. The `IsToggled` property is the state of the switch, and the `Toggled` event is fired when the state changes.

[API Documentation](#)



[C# code for this page / XAML page](#)

## DatePicker

**DatePicker** allows the user to select a date with the platform date picker. Set a range of allowable dates with the `MinimumDate` and `MaximumDate` properties. The `Date` property is the selected date, and the `SelectedDateChanged` event is fired when that property changes.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

## TimePicker

**TimePicker** allows the user to select a time with the platform time picker. The `Time` property is the selected time. An application can monitor changes in the `Time` property by installing a handler for the `PropertyChanged` event.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

## Views for editing text

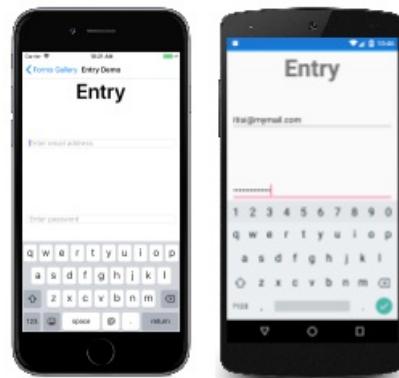
These two classes derive from the `InputView` class, which defines the `Keyboard` property.

### Entry

**Entry** allows the user to enter and edit a single line of text. The text is available as the **Text** property, and the **TextChanged** and **Completed** events are fired when the text changes or the user signals completion by tapping the enter key.

Use an **Editor** for entering and editing multiple lines of text.

[API Documentation](#) / [Guide](#) / [Sample](#)



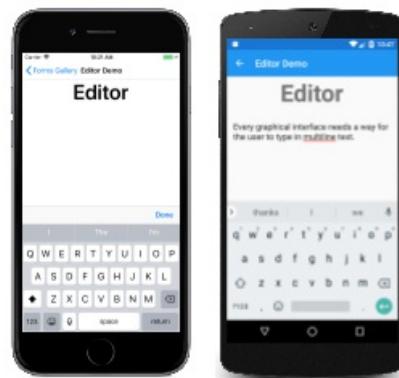
[C# code for this page](#) / [XAML page](#)

## Editor

**Editor** allows the user to enter and edit multiple lines of text. The text is available as the **Text** property, and the **TextChanged** and **Completed** events are fired when the text changes or the user signals completion.

Use an **Entry** view for entering and editing a single line of text.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#)

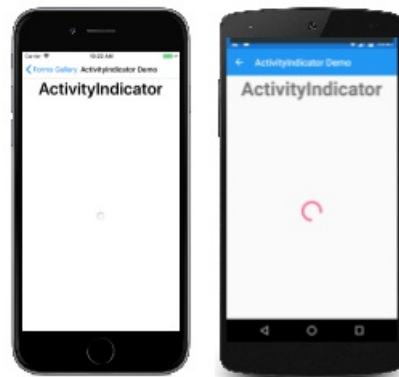
# Views to indicate activity

## ActivityIndicator

**ActivityIndicator** uses an animation to show that the application is engaged in a lengthy activity without giving any indication of progress. The **IsRunning** property controls the animation.

If the activity's progress is known, use a **ProgressBar** instead.

[API Documentation](#)



[C# code for this page](#) / [XAML page](#)

## ProgressBar

`ProgressBar` uses an animation to show that the application is progressing through a lengthy activity. Set the `Progress` property to values between 0 and 1 to indicate the progress.

If the activity's progress is not known, use an `ActivityIndicator` instead.

[API Documentation](#)



[C# code for this page / XAML page with code-behind](#)

## Views that display collections

### Picker

`Picker` displays a selected item from a list of text strings, and allows selecting that item when the view is tapped. Set the `Items` property to a list of strings, or the `ItemsSource` property to a collection of objects. The `SelectedIndexChanged` event is fired when an item is selected.

The `Picker` displays the list of items only when it's selected. Use a `ListView` or `TableView` for a scrollable list that remains on the page.

[API Documentation / Guide / Sample](#)

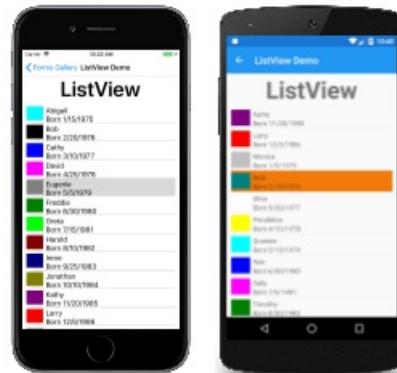


[C# code for this page / XAML page with code-behind](#)

### ListView

`ListView` derives from `ItemsView[Cell]` and displays a scrollable list of selectable data items. Set the `ItemsSource` property to a collection of objects, and set the `ItemTemplate` property to a `DataTemplate` object describing how the items are to be formatted. The `ItemSelected` event signals that a selection has been made, which is available as the `SelectedItem` property.

[API Documentation / Guide / Sample](#)



[C# code for this page / XAML page](#)

### TableView

`TableView` displays a list of rows of type `Cell` with optional headers and subheaders. Set the `Root` property to an object of type `TableRoot`, and add `TableSection` objects to that `TableRoot`. Each `TableSection` is a collection of `Cell` objects.

[API Documentation](#) / [Guide](#) / [Sample](#)



[C# code for this page](#) / [XAML page](#)

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms Cells

11/11/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms cells can be added to `ListView`s and `TableViews`.

A `cell` is a specialized element used for items in a table and describes how each item in a list should be rendered.

The `Cell` class derives from `Element`, from which `VisualElement` also derives. A cell is not itself a visual element; it is instead a template for creating a visual element.

`Cell` is used exclusively with `ListView` and `TableView` controls. To learn how to use and customize cells, refer to the `ListView` and `TableView` documentation.

## Cells

Xamarin.Forms supports the following cell types:

### TextCell

A `TextCell` displays one or two text strings. Set the `Text` property and, optionally, the `Detail` property to these text strings.

[API Documentation / Guide](#)



[C# code for this page / XAML page](#)

### ImageCell

The `ImageCell` displays the same information as `TextCell` but includes a bitmap that you set with the [Source](#) property.

[API Documentation / Guide](#)



[C# code for this page / XAML page](#)

### SwitchCell

The `SwitchCell` contains text set with the `Text` property and on/off switch initially set with the Boolean `On` property. Handle the `OnChanged` event to be notified when the `On` property changes.

[API Documentation / Guide](#)

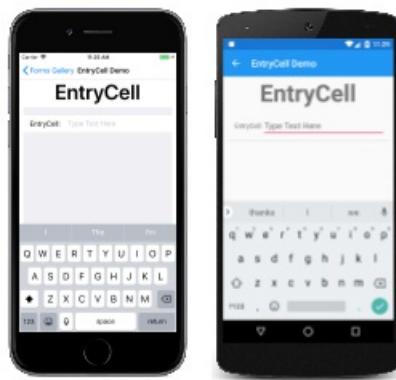


[C# code for this page / XAML page](#)

## EntryCell

The `EntryCell` defines a `Label` property that identifies the cell and a single line of editable text in the `Text` property. Handle the `Completed` event to be notified when the user has completed the text entry.

[API Documentation / Guide](#)



[C# code for this page / XAML page](#)

## Related Links

- [Introduction To Xamarin.Forms](#)
- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

# Xamarin.Forms DataPages

6/8/2018 • 2 minutes to read • [Edit Online](#)



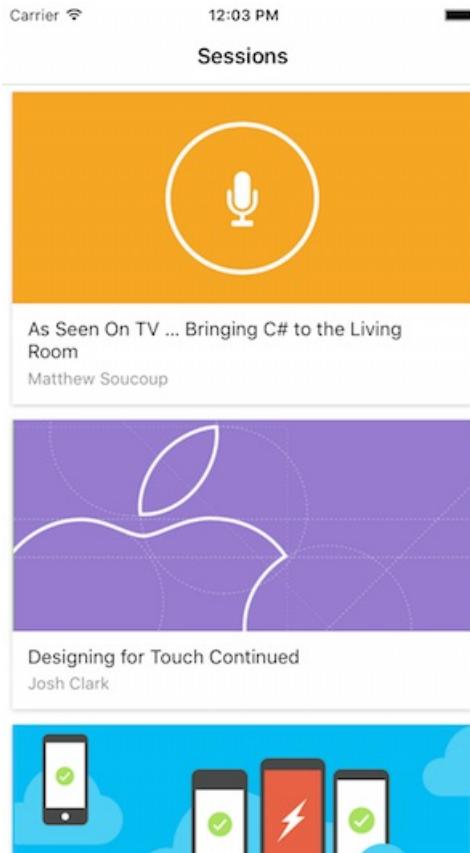
## IMPORTANT

DataPages requires a [Xamarin.Forms Theme](#) reference to render.

Xamarin.Forms DataPages were announced at Evolve 2016 and are available as a preview for customers to try and provide feedback.

DataPages provide an API to quickly and easily bind a data source to pre-built views. List items and detail pages will automatically render the data, and can be customized using themes.

To see how the Evolve keynote demo works, check out the [getting started guide](#).



## Introduction

Data sources and the associated data pages allow developers to quickly and easily consume a supported data source and render it using built-in UI scaffolding that can be customized with themes.

DataPages are added to a Xamarin.Forms application by including the **Xamarin.Forms.Pages** Nuget package.

## Data Sources

The Preview has some prebuilt data sources available for use:

- **JsonDataSource**
- **AzureDataSource** (separate Nuget)
- **AzureEasyTableDataSource** (separate Nuget)

See the [getting started guide](#) for an example using a `JsonDataSource`.

## Pages & Controls

The following pages and controls are included to allow easy binding to the supplied data sources:

- **ListDataPage** – see the [getting started example](#).
- **DirectoryPage** – a list with grouping enabled.
- **PersonDetailPage** – a single data item view customized for a specific object type (a contact entry).
- **DataView** – a view to expose data from the source in a generic fashion.
- **CardView** – a styled view that contains an image, title text, and description text.
- **HeroImage** – an image-rendering view.
- **ListItem** – a pre-built view with a layout similar to native iOS and Android list items.

See the [DataPages controls reference](#) for examples.

## Under the Hood

A Xamarin.Forms data source adheres to the `IDataSource` interface.

The Xamarin.Forms infrastructure interacts with a data source through the following properties:

- `Data` – a read-only list of data items that can be displayed.
- `IsLoading` – a boolean indicating whether the data is loaded and available for rendering.
- `[key]` – an indexer to retrieve elements.

There are two methods `MaskKey` and `UnmaskKey` that can be used to hide (or show) data item properties (ie. prevent them from being rendered). The key corresponds to the a named property on the data item object.

# Getting Started with DataPages

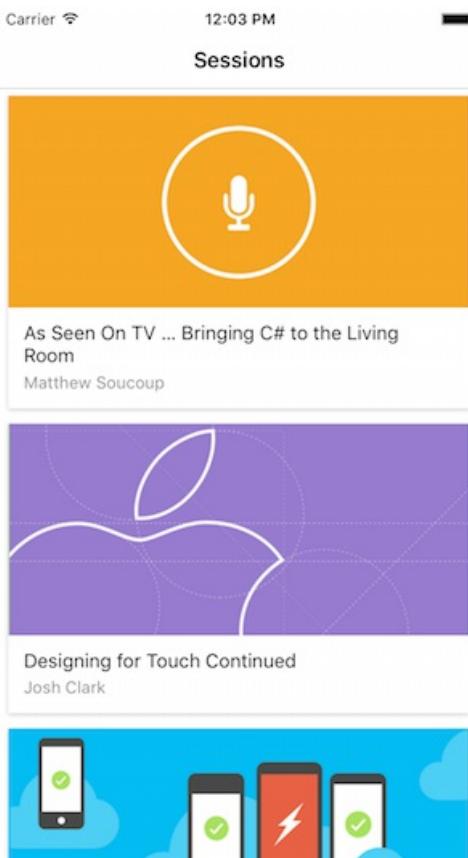
6/8/2018 • 4 minutes to read • [Edit Online](#)



## IMPORTANT

DataPages requires a [Xamarin.Forms Theme](#) reference to render.

To get started building a simple data-driven page using the DataPages Preview, follow the steps below. This demo uses a hardcoded style ("Events") in the Preview builds that only works with the specific JSON format in the code.



## 1. Add NuGet Packages

Add these Nuget packages to your Xamarin.Forms .NET Standard library and application projects:

- `Xamarin.Forms.Pages`
- `Xamarin.Forms.Theme.Base`
- A theme implementation Nuget (eg. `Xamarin.Forms.Themes.Light`)

## 2. Add Theme Reference

In the `App.xaml` file, add a custom `xmlns:mytheme` for the theme and ensure the theme is merged into the application's resource dictionary:

```

<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:mytheme="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Light"
    x:Class="DataPagesDemo.App">
    <Application.Resources>
        <ResourceDictionary MergedWith="mytheme:LightThemeResources" />
    </Application.Resources>
</Application>

```

**IMPORTANT:** You should also follow the steps to [load theme assemblies \(below\)](#) by adding some boilerplate code to the iOS `AppDelegate` and Android `MainActivity`. This will be improved in a future preview release.

### 3. Add a XAML Page

Add a new XAML page to the Xamarin.Forms application, and *change the base class* from `ContentPage` to `Xamarin.Forms.Pages.ListDataPage`. This has to be done in both the C# and the XAML:

#### C# file

```

public partial class SessionDataPage : Xamarin.Forms.Pages.ListDataPage // was ContentPage
{
    public SessionDataPage ()
    {
        InitializeComponent ();
    }
}

```

#### XAML file

In addition to changing the root element to `<p:ListDataPage>` the custom namespace for `xmlns:p` must also be added:

```

<?xml version="1.0" encoding="UTF-8"?>
<p:ListDataPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:p="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
    x:Class="DataPagesDemo.SessionDataPage">

    <ContentPage.Content></ContentPage.Content>

</p:ListDataPage>

```

#### Application subclass

Change the `App` class constructor so that the `MainPage` is set to a `NavigationPage` containing the new `SessionDataPage`. A navigation page *must* be used.

```

MainPage = new NavigationPage (new SessionDataPage ());

```

### 3. Add the DataSource

Delete the `Content` element and replace it with a `p:ListDataPage.DataSource` to populate the page with data. In the example below a remote Json data file is being loaded from a URL.

**Note:** the preview *requires* a `StyleClass` attribute to provide rendering hints for the data source. The `StyleClass="Events"` refers to a layout that is predefined in the preview and contains styles *hardcoded* to match

the JSON data source being used.

```
<?xml version="1.0" encoding="UTF-8"?>
<p:ListDataPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:p="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
    x:Class="DataPagesDemo.SessionDataPage"
    Title="Sessions" StyleClass="Events">

    <p:ListDataPage.DataSource>
        <p:JsonDataSource Source="http://demo3143189.mockable.io/sessions" />
    </p:ListDataPage.DataSource>

</p:ListDataPage>
```

## JSON data

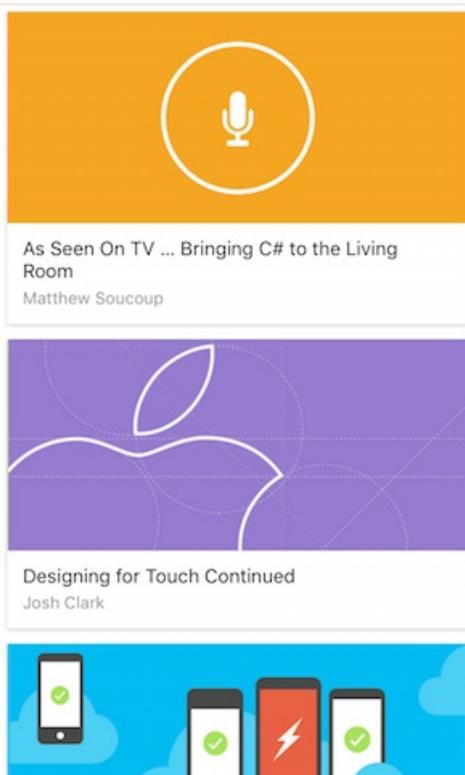
An example of the JSON data from the [demo source](#) is shown below:

```
[{
    "end": "2016-04-27T18:00:00Z",
    "start": "2016-04-27T17:15:00Z",
    "abstract": "The new Apple TV has been released, and YOU can be one of the first developers to write apps for it. To make things even better, you can build these apps in C#! This session will introduce the basics of how to create a tvOS app with Xamarin, including: differences between tvOS and iOS APIs, TV user interface best practices, responding to user input, as well as the capabilities and limitations of building apps for a television. Grab some popcorn--this is going to be good!",
    "title": "As Seen On TV ... Bringing C# to the Living Room",
    "presenter": "Matthew Soucoup",
    "biography": "Matthew is a Xamarin MVP and Certified Xamarin Developer from Madison, WI. He founded his company Code Mill Technologies and started the Madison Mobile .Net Developers Group. Matt regularly speaks on .Net and Xamarin development at user groups, code camps and conferences throughout the Midwest. Matt gardens hot peppers, rides bikes, and loves Wisconsin micro-brews and cheese.",
    "image": "http://i.imgur.com/ASj60DP.jpg",
    "avatar": "http://i.imgur.com/ASj60DP.jpg",
    "room": "Crick"
}]
```

## 4. Run!

The above steps should result in a working data page:

## Sessions



This works because the pre-built style "**Events**" exists in the Light Theme Nuget package and has styles defined that match the data source (eg. "title", "image", "presenter").

The "Events" `StyleClass` is built to display the `ListDataPage` control with a custom `CardView` control that is defined in `Xamarin.Forms.Pages`. The `CardView` control has three properties: `ImageSource`, `Text`, and `Detail`. The theme is hardcoded to bind the datasource's three fields (from the JSON file) to these properties for display.

## 5. Customize

The inherited style can be overridden by specifying a template and using data source bindings. The XAML below declares a custom template for each row using the new `ListItemControl` and `{p:DataSourceBinding}` syntax which is included in the **Xamarin.Forms.Pages** Nuget:

```
<p>ListDataPage.DefaultItemTemplate>
<DataTemplate>
<ViewCell>
<p>ListItemControl
    Title="{p:DataSourceBinding title}"
    Detail="{p:DataSourceBinding room}"
    ImageSource="{p:DataSourceBinding image}"
    DataSource="{Binding Value}"
    HeightRequest="90"
  >
</p>ListItemControl>
</ViewCell>
</DataTemplate>
</p>ListDataPage.DefaultItemTemplate>
```

By providing a `DataTemplate` this code overrides the `StyleClass` and instead uses the default layout for a `ListItemControl`.

## Sessions



As Seen On TV ... Bringing C# to the Living Room  
Crick



Designing for Touch Continued  
Linnaeus



By Our Own Devices: Will Robots Take Our Jobs?  
Watson



Google Fit, Android Wear, and Xamarin  
Franklin



Cross-Platform Media in Xamarin  
Watson



Best Practices for Effective iOS Memory Management  
Watson



Creating Custom Layouts in Xamarin.Forms

Developers that prefer C# to XAML can create data source bindings too (remember to include a

```
using Xamarin.Forms.Pages; statement):
```

```
SetBinding (TitleProperty, new DataSourceBinding ("title"));
```

It's a little more work to create themes from scratch (see the [Themes guide](#)) but future preview releases will make this easier to do.

## Troubleshooting

### Could not load file or assembly 'Xamarin.Forms.Theme.Light' or one of its dependencies

In the preview release, themes may not be able to load at runtime. Add the code shown below into the relevant projects to fix this error.

#### iOS

In the **AppDelegate.cs** add the following lines after `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.iOS.UnderlineEffect);
```

#### Android

In the **MainActivity.cs** add the following lines after `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);  
x = typeof(Xamarin.Forms.Themes.LightThemeResources);  
x = typeof(Xamarin.Forms.Themes.Android.UnderlineEffect);
```

## Related Links

- [DataPagesDemo sample](#)

# DataPages Controls Reference

6/8/2018 • 4 minutes to read • [Edit Online](#)



## IMPORTANT

DataPages requires a [Xamarin.Forms Theme](#) reference to render.

The Xamarin.Forms DataPages Nuget includes a number of controls that can take advantage of data source binding.

To use these controls in XAML, ensure the namespace has been included, for example see the `xmlns:pages` declaration below:

```
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
    x:Class="DataPagesDemo.Detail">
```

The examples below include `DynamicResource` references which would need to exist in the project's resources dictionary to work. There is also an example of how to build a [custom control](#)

## Built-in Controls

- [HeroImage](#)
- [ListItem](#)

### **HeroImage**

The `HeroImage` control has four properties:

- `Text`
- `Detail`
- `ImageSource`
- `Aspect`

```
<pages:HeroImage
    ImageSource="{ DynamicResource HeroImageImage }"
    Text="Keith Ballinger"
    Detail="Xamarin"
    />
```

## Android



## iOS



## ListItem

The `ListItem` control's layout is similar to native iOS and Android list or table rows, however it can also be used as a regular view. In the example code below it is shown hosted inside a `StackLayout`, but it can also be used in data-bound scrolling list controls.

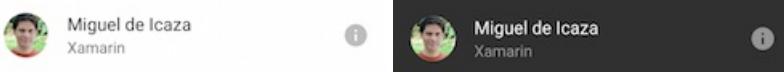
There are five properties:

- Title
- Detail
- ImageSource
- PlaceholderImageSource
- Aspect

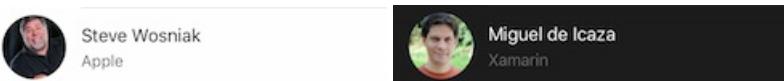
```
<StackLayout Spacing="0">
    <pages:ListItemControl
        Detail="Xamarin"
        ImageSource="{ DynamicResource UserImage }"
        Title="Miguel de Icaza"
        PlaceholderImageSource="{ DynamicResource IconImage }"
    />
```

These screenshots show the `ListItem` on iOS and Android platforms using both the Light and Dark themes:

## Android



## iOS



## Custom Control Example

The goal of this custom `CardView` control is to resemble the native Android CardView.

It will contain three properties:

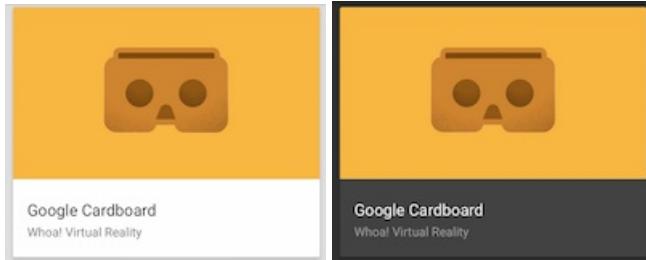
- Text
- Detail
- ImageSource

The goal is a custom control that will look like the code below (note that a custom `xmlns:local` is required that references the current assembly):

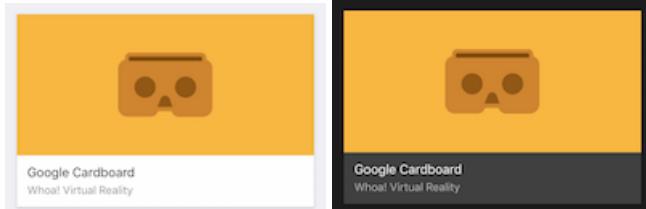
```
<local:CardView
    ImageSource="{ DynamicResource CardViewImage }"
    Text="CardView Text"
    Detail="CardView Detail"
/>
```

It should look like the screenshots below using colors corresponding to the built-in Light and Dark themes:

## Android



## iOS



## Building the Custom CardView

1. [DataView subclass](#)
2. [Define Font, Layout, and Margins](#)
3. [Create Styles for the Control's Children](#)
4. [Create the Control Layout Template](#)
5. [Add the Theme-specific Resources](#)
6. [Set the ControlTemplate for the CardView class](#)
7. [Add the Control to a Page](#)

### 1. DataView Subclass

The C# subclass of `DataView` defines the bindable properties for the control.

```

public class CardView : DataView
{
    public static readonly BindableProperty TextProperty =
        BindableProperty.Create ("Text", typeof (string), typeof (CardView), null, BindingMode.TwoWay);

    public string Text
    {
        get { return (string)GetValue (TextProperty); }
        set { SetValue (TextProperty, value); }
    }

    public static readonly BindableProperty DetailProperty =
        BindableProperty.Create ("Detail", typeof (string), typeof (CardView), null, BindingMode.TwoWay);

    public string Detail
    {
        get { return (string)GetValue (DetailProperty); }
        set { SetValue (DetailProperty, value); }
    }

    public static readonly BindableProperty ImageSourceProperty =
        BindableProperty.Create ("ImageSource", typeof (ImageSource), typeof (CardView), null,
BindingMode.TwoWay);

    public ImageSource ImageSource
    {
        get { return (ImageSource)GetValue (ImageSourceProperty); }
        set { SetValue (ImageSourceProperty, value); }
    }

    public CardView()
    {
    }
}

```

## 2. Define Font, Layout, and Margins

The control designer would figure out these values as part of the user-interface design for the custom control. Where platform-specific specifications are required, the `onPlatform` element is used.

Note that some values refer to `StaticResource S` – these will be defined in [step 5](#).

```

<!-- CARDVIEW FONT SIZES -->
<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewTextFontSize">
    <On Platform="iOS, Android" Value="15" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewDetailFontSize">
    <On Platform="iOS, Android" Value="13" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"     x:Key="CardViewTextTextColor">
    <On Platform="iOS" Value="{StaticResource iOSCardViewTextTextColor}" />
    <On Platform="Android" Value="{StaticResource AndroidCardViewTextTextColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="Thickness"   x:Key="CardViewTextMargin">
    <On Platform="iOS" Value="12,10,12,4" />
    <On Platform="Android" Value="20,0,20,5" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"     x:Key="CardViewDetailTextColor">
    <On Platform="iOS" Value="{StaticResource iOSCardViewDetailTextColor}" />
    <On Platform="Android" Value="{StaticResource AndroidCardViewDetailTextColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="Thickness"   x:Key="CardViewDetailMargin">
    <On Platform="iOS" Value="12,0,10,12" />
    <On Platform="Android" Value="20,0,20,20" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"     x:Key="CardViewBackgroundColor">
    <On Platform="iOS" Value="{StaticResource iOSCardViewBackgroundColor}" />
    <On Platform="Android" Value="{StaticResource AndroidCardViewBackgroundColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewShadowSize">
    <On Platform="iOS" Value="2" />
    <On Platform="Android" Value="5" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewCornerRadius">
    <On Platform="iOS" Value="0" />
    <On Platform="Android" Value="4" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"     x:Key="CardViewShadowColor">
    <On Platform="iOS, Android" Value="#CDCDD1" />
</OnPlatform>

```

### 3. Create Styles for the Control's Children

Reference all the elements defined above to create the children that will be used in the custom control:

```

<!-- EXPLICIT STYLES (will be Classes) -->
<Style TargetType="Label" x:Key="CardViewTextStyle">
    <Setter Property="FontSize" Value="{ StaticResource CardViewTextFontSize }" />
    <Setter Property="TextColor" Value="{ StaticResource CardViewTextTextColor }" />
    <Setter Property="HorizontalOptions" Value="Start" />
    <Setter Property="Margin" Value="{ StaticResource CardViewTextMargin }" />
    <Setter Property="HorizontalTextAlignment" Value="Start" />
</Style>

<Style TargetType="Label" x:Key="CardViewDetailStyle">
    <Setter Property="HorizontalTextAlignment" Value="Start" />
    <Setter Property="TextColor" Value="{ StaticResource CardViewDetailTextColor }" />
    <Setter Property="FontSize" Value="{ StaticResource CardViewDetailFontSize }" />
    <Setter Property="HorizontalOptions" Value="Start" />
    <Setter Property="Margin" Value="{ StaticResource CardViewDetailMargin }" />
</Style>

<Style TargetType="Image" x:Key="CardViewImageImageStyle">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="Center" />
    <Setter Property="WidthRequest" Value="220"/>
    <Setter Property="HeightRequest" Value="165"/>
</Style>

```

#### 4. Create the Control Layout Template

The visual design of the custom control is explicitly declared in the control template, using the resources defined above:

```

<!-- CARDVIEW -->
<ControlTemplate x:Key="CardViewControlControlTemplate">
    <StackLayout
        Spacing="0"
        BackgroundColor="{ TemplateBinding BackgroundColor }"
    >

        <!-- CARDVIEW IMAGE -->
        <Image
            Source="{ TemplateBinding ImageSource }"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="StartAndExpand"
            Aspect="AspectFill"
            Style="{ StaticResource CardViewImageImageStyle }"
        />

        <!-- CARDVIEW TEXT -->
        <Label
            Text="{ TemplateBinding Text }"
            LineBreakMode="WordWrap"
            VerticalOptions="End"
            Style="{ StaticResource CardViewTextStyle }"
        />

        <!-- CARDVIEW DETAIL -->
        <Label
            Text="{ TemplateBinding Detail }"
            LineBreakMode="WordWrap"
            VerticalOptions="End"
            Style="{ StaticResource CardViewDetailStyle }" />

    </StackLayout>
</ControlTemplate>

```

## 5. Add the Theme-specific Resources

Because this is a custom control, add the resources that match the theme you are using the resource dictionary:

### Light Theme Colors

```
<Color x:Key="iOSCardViewBackgroundColor">#FFFFFF</Color>
<Color x:Key="AndroidCardViewBackgroundColor">#FFFFFF</Color>

<Color x:Key="AndroidCardViewTextTextColor">#030303</Color>
<Color x:Key="iOSCardViewTextTextColor">#030303</Color>

<Color x:Key="AndroidCardViewDetailTextColor">#8F8E94</Color>
<Color x:Key="iOSCardViewDetailTextColor">#8F8E94</Color>
```

### Dark Theme Colors

```
<!-- CARD VIEW COLORS -->
<Color x:Key="iOSCardViewBackgroundColor">#404040</Color>
<Color x:Key="AndroidCardViewBackgroundColor">#404040</Color>

<Color x:Key="AndroidCardViewTextTextColor">#FFFFFF</Color>
<Color x:Key="iOSCardViewTextTextColor">#FFFFFF</Color>

<Color x:Key="AndroidCardViewDetailTextColor">#B5B4B9</Color>
<Color x:Key="iOSCardViewDetailTextColor">#B5B4B9</Color>
```

## 6. Set the ControlTemplate for the CardView class

Finally, ensure the C# class created in [step 1](#) uses the control template defined in [step 4](#) using a `Style` `Setter` element

```
<Style TargetType="local:CardView">
    <Setter Property="ControlTemplate" Value="{ StaticResource CardViewControlControlTemplate }" />
    ... some custom styling omitted
    <Setter Property="BackgroundColor" Value="{ StaticResource CardViewBackgroundColor }" />
</Style>
```

## 7. Add the Control to a Page

The `CardView` control can now be added to a page. The example below shows it hosted in a `StackLayout`:

```
<StackLayout Spacing="0">
    <local:CardView
        Margin="12,6"
        ImageSource="{ DynamicResource CardViewImage }"
        Text="CardView Text"
        Detail="CardView Detail"
    />
</StackLayout>
```

# Xamarin.Forms DatePicker

11/20/2018 • 5 minutes to read • [Edit Online](#)

A *Xamarin.Forms* view that allows the user to select a date.

The *Xamarin.Forms* `DatePicker` invokes the platform's date-picker control and allows the user to select a date. `DatePicker` defines eight properties:

- `MinimumDate` of type `DateTime`, which defaults to the first day of the year 1900.
- `MaximumDate` of type `DateTime`, which defaults to the last day of the year 2100.
- `Date` of type `DateTime`, the selected date, which defaults to the value `DateTime.Today`.
- `Format` of type `string`, a [standard](#) or [custom](#) .NET formatting string, which defaults to "D", the long date pattern.
- `TextColor` of type `Color`, the color used to display the selected date, which defaults to `Color.Default`.
- `FontAttributes` of type `FontAttributes`, which defaults to `FontAttributes.None`.
- `FontFamily` of type `string`, which defaults to `null`.
- `FontSize` of type `double`, which defaults to -1.0.

The `DatePicker` fires a `DateSelected` event when the user selects a date.

## WARNING

When setting `MinimumDate` and `MaximumDate`, make sure that `MinimumDate` is always less than or equal to `MaximumDate`. Otherwise, `DatePicker` will raise an exception.

Internally, the `DatePicker` ensures that `Date` is between `MinimumDate` and `MaximumDate`, inclusive. If `MinimumDate` or `MaximumDate` is set so that `Date` is not between them, `DatePicker` will adjust the value of `Date`.

All eight properties are backed by `BindableProperty` objects, which means that they can be styled, and the properties can be targets of data bindings. The `Date` property has a default binding mode of `BindingMode.TwoWay`, which means that it can be a target of a data binding in an application that uses the [Model-View-ViewModel \(MVVM\)](#) architecture.

## Initializing the DateTime properties

In code, you can initialize the `MinimumDate`, `MaximumDate`, and `Date` properties to values of type `DateTime`:

```
DatePicker datePicker = new DatePicker
{
    MinimumDate = new DateTime(2018, 1, 1),
    MaximumDate = new DateTime(2018, 12, 31),
    Date = new DateTime(2018, 6, 21)
};
```

When a `DateTime` value is specified in XAML, the XAML parser uses the `DateTime.Parse` method with a `CultureInfo.InvariantCulture` argument to convert the string to a `DateTime` value. The dates must be specified in a precise format: two-digit months, two-digit days, and four-digit years separated by slashes:

```
<DatePicker MinimumDate="01/01/2018"  
           MaximumDate="12/31/2018"  
           Date="06/21/2018" />
```

If the `BindingContext` property of `DatePicker` is set to an instance of a ViewModel containing properties of type `DateTime` named `MinDate`, `MaxDate`, and `SelectedDate` (for example), you can instantiate the `DatePicker` like this:

```
<DatePicker MinimumDate="{Binding MinDate}"  
           MaximumDate="{Binding MaxDate}"  
           Date="{Binding SelectedDate}" />
```

In this example, all three properties are initialized to the corresponding properties in the ViewModel. Because the `Date` property has a binding mode of `TwoWay`, any new date that the user selects is automatically reflected in the ViewModel.

If the `DatePicker` does not contain a binding on its `Date` property, an application should attach a handler to the `Selected` event to be informed when the user selects a new date.

For information about setting font properties, see [Fonts](#).

## DatePicker and layout

It's possible to use an unconstrained horizontal layout option such as `Center`, `Start`, or `End` with `DatePicker`:

```
<DatePicker ...  
             HorizontalOptions="Center"  
             ... />
```

However, this is not recommended. Depending on the setting of the `Format` property, selected dates might require different display widths. For example, the "D" format string causes `DateTime` to display dates in a long format, and "Wednesday, September 12, 2018" requires a greater display width than "Friday, May 4, 2018". Depending on the platform, this difference might cause the `DateTime` view to change width in layout, or for the display to be truncated.

### TIP

It's best to use the default `HorizontalOptions` setting of `Fill` with `DatePicker`, and not to use a width of `Auto` when putting `DatePicker` in a `Grid` cell.

## DatePicker in an application

The [DaysBetweenDates](#) sample includes two `DatePicker` views on its page. These can be used to select two dates, and the program calculates the number of days between those dates. The program doesn't change the settings of the `MinimumDate` and `MaximumDate` properties, so the two dates must be between 1900 and 2100.

Here's the XAML file:

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DaysBetweenDates"
    x:Class="DaysBetweenDates.MainPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <StackLayout Margin="10">
        <Label Text="Days Between Dates"
            Style="{DynamicResource TitleStyle}"
            Margin="0, 20"
            HorizontalTextAlignment="Center" />

        <Label Text="Start Date:" />

        <DatePicker x:Name="startDatePicker"
            Format="D"
            Margin="30, 0, 0, 30"
            DateSelected="OnDateSelected" />

        <Label Text="End Date:" />

        <DatePicker x:Name="endDatePicker"
            MinimumDate="{Binding Source={x:Reference startDatePicker},
                Path=Date}"
            Format="D"
            Margin="30, 0, 0, 30"
            DateSelected="OnDateSelected" />

        <StackLayout Orientation="Horizontal"
            Margin="0, 0, 0, 30">
            <Label Text="Include both days in total: "
                VerticalOptions="Center" />
            <Switch x:Name="includeSwitch"
                Toggled="OnSwitchToggled" />
        </StackLayout>

        <Label x:Name="resultLabel"
            FontAttributes="Bold"
            HorizontalTextAlignment="Center" />
    </StackLayout>
</ContentPage>

```

Each `DatePicker` is assigned a `Format` property of "D" for a long date format. Notice also that the `endDatePicker` object has a binding that targets its `MinimumDate` property. The binding source is the selected `Date` property of the `startDatePicker` object. This ensures that the end date is always later than or equal to the start date. In addition to the two `DatePicker` objects, a `Switch` is labeled "Include both days in total".

The two `DatePicker` views have handlers attached to the `DateSelected` event, and the `Switch` has a handler attached to its `Toggled` event. These event handlers are in the code-behind file and trigger a new calculation of the days between the two dates:

```

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    void OnDateSelected(object sender, DateChangedEventArgs args)
    {
        Recalculate();
    }

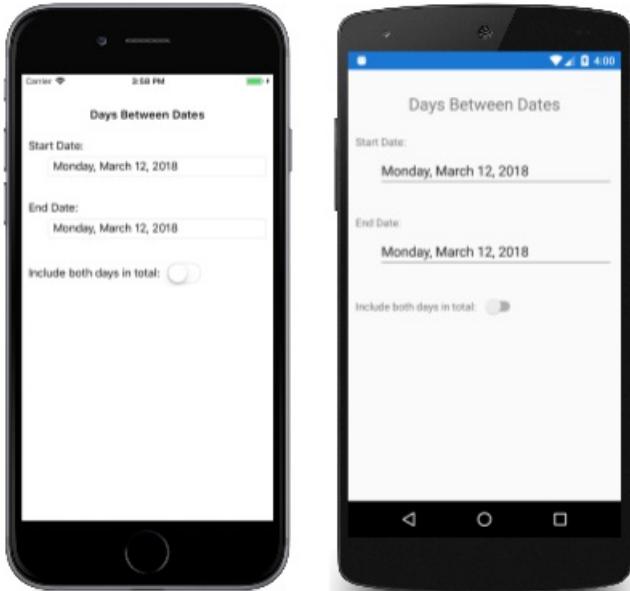
    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        Recalculate();
    }

    void Recalculate()
    {
        TimeSpan timeSpan = endDatePicker.Date - startDatePicker.Date +
            (includeSwitch.IsToggled ? TimeSpan.FromDays(1) : TimeSpan.Zero);

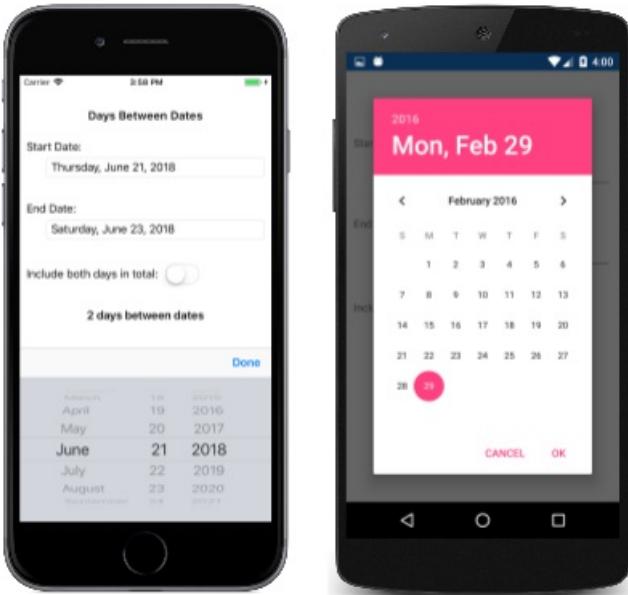
        resultLabel.Text = String.Format("{0} day{1} between dates",
                                         timeSpan.Days, timeSpan.Days == 1 ? "" : "s");
    }
}

```

When the sample is first run, both `DatePicker` views are initialized to today's date. The following screenshot shows the program running on iOS, Android, and the Universal Windows Platform:



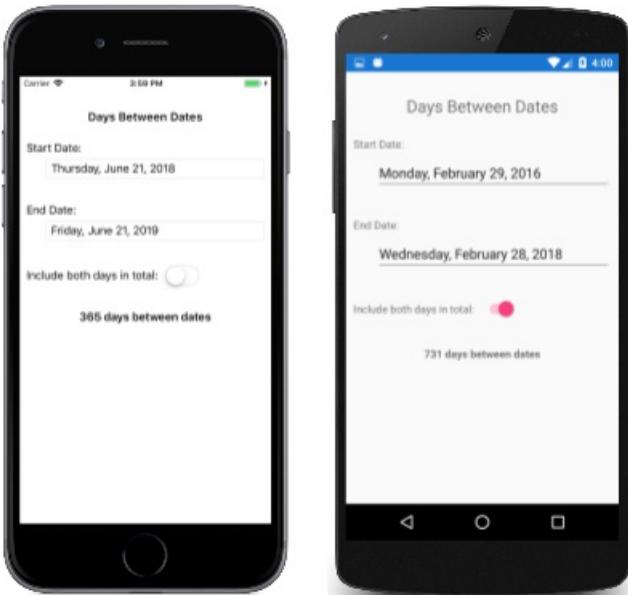
Tapping either of the `DatePicker` displays invokes the platform date picker. The platforms implement this date picker in very different ways, but each approach is familiar to users of that platform:



### TIP

On Android, the `DatePicker` dialog can be customized by overriding the `CreateDatePickerDialog` method in a custom renderer. This allows, for example, additional buttons to be added to the dialog.

After two dates are selected, the application displays the number of days between those dates:



## Related links

- [DaysBetweenDates sample](#)
- [DatePicker API](#)

# SkiaSharp Graphics in Xamarin.Forms

10/3/2018 • 2 minutes to read • [Edit Online](#)

*Use SkiaSharp for 2D graphics in your Xamarin.Forms applications*

SkiaSharp is a 2D graphics system for .NET and C# powered by the open-source Skia graphics engine that is used extensively in Google products. You can use SkiaSharp in your Xamarin.Forms applications to draw 2D vector graphics, bitmaps, and text. See the [2D Drawing](#) guide for more general information about the SkiaSharp library and some other tutorials.

This guide assumes that you are familiar with Xamarin.Forms programming.

## Webinar: SkiaSharp for Xamarin.Forms

## SkiaSharp Preliminaries

SkiaSharp for Xamarin.Forms is packaged as a NuGet package. After you've created a Xamarin.Forms solution in Visual Studio or Visual Studio for Mac, you can use the NuGet package manager to search for the **SkiaSharp.Views.Forms** package and add it to your solution. If you check the **References** section of each project after adding SkiaSharp, you can see that various **SkiaSharp** libraries have been added to each of the projects in the solution.

If your Xamarin.Forms application targets iOS, use the project properties page to change the minimum deployment target to iOS 8.0.

In any C# page that uses SkiaSharp you'll want to include a `using` directive for the `SkiaSharp` namespace, which encompasses all the SkiaSharp classes, structures, and enumerations that you'll use in your graphics programming. You'll also want a `using` directive for the `SkiaSharp.Views.Forms` namespace for the classes specific to Xamarin.Forms. This is a much smaller namespace, with the most important class being `SKCanvasView`. This class derives from the Xamarin.Forms `View` class and hosts your SkiaSharp graphics output.

### IMPORTANT

The `SkiaSharp.Views.Forms` namespace also contains an `SKGLView` class that derives from `View` but uses OpenGL for rendering graphics. For purposes of simplicity, this guide restricts itself to `SKCanvasView`, but using `SKGLView` instead is quite similar.

## SkiaSharp Drawing Basics

Some of the simplest graphics figures you can draw with SkiaSharp are circles, ovals, and rectangles. In displaying these figures, you will learn about SkiaSharp coordinates, sizes, and colors. The display of text and bitmaps is more complex, but these articles also introduce those techniques.

## SkiaSharp Lines and Paths

A graphics path is a series of connected straight lines and curves. Paths can be stroked, filled, or both. This article encompasses many aspects of line drawing, including stroke ends and joins, and dashed and dotted lines, but stops short of curve geometries.

## SkiaSharp Transforms

Transforms allow graphics objects to be uniformly translated, scaled, rotated, or skewed. This article also shows how you can use a standard 3-by-3 transform matrix for creating non-affine transforms and applying transforms to paths.

## SkiaSharp Curves and Paths

The exploration of paths continues with adding curves to a path objects, and exploiting other powerful path features. You'll see how you can specify an entire path in a concise text string, how to use path effects, and how to dig into path internals.

## SkiaSharp Bitmaps

Bitmaps are rectangular arrays of bits corresponding to the pixels of a display device. This series of articles shows how to load, save, display, create, draw on, animate, and access the bits of SkiaSharp bitmaps.

## SkiaSharp Effects

Effects are properties that alter the normal display of graphics, including linear and circular gradients, bitmap tiling, blend modes, blur, and others.

## Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [SkiaSharp with Xamarin.Forms Webinar \(video\)](#)

# Images in Xamarin.Forms

11/20/2018 • 11 minutes to read • [Edit Online](#)

*Images can be shared across platforms with Xamarin.Forms, they can be loaded specifically for each platform, or they can be downloaded for display.*

Images are a crucial part of application navigation, usability, and branding. Xamarin.Forms applications need to be able to share images across all platforms, but also potentially display different images on each platform.

Platform-specific images are also required for icons and splash screens; these need to be configured on a per-platform basis.

## Displaying Images

Xamarin.Forms uses the `Image` view to display images on a page. It has two important properties:

- `Source` - An `ImageSource` instance, either File, Uri or Resource, which sets the image to display.
- `Aspect` - How to size the image within the bounds it is being displayed within (whether to stretch, crop or letterbox).

`ImageSource` instances can be obtained using static methods for each type of image source:

- `FromFile` - Requires a filename or filepath that can be resolved on each platform.
- `FromUri` - Requires a Uri object, eg. `new Uri("http://server.com/image.jpg")`.
- `FromResource` - Requires a resource identifier to an image file embedded in the application or .NET Standard library project, with a **Build Action:EmbeddedResource**.
- `FromStream` - Requires a stream that supplies image data.

The `Aspect` property determines how the image will be scaled to fit the display area:

- `Fill` - Stretches the image to completely and exactly fill the display area. This may result in the image being distorted.
- `AspectFill` - Clips the image so that it fills the display area while preserving the aspect (ie. no distortion).
- `AspectFit` - Letterboxes the image (if required) so that the entire image fits into the display area, with blank space added to the top/bottom or sides depending on whether the image is wide or tall.

Images can be loaded from a [local file](#), an [embedded resource](#), or [downloaded](#).

## Local Images

Image files can be added to each application project and referenced from Xamarin.Forms shared code. This method of distributing images is required when images are platform-specific, such as when using different resolutions on different platforms, or slightly different designs.

To use a single image across all apps, *the same filename must be used on every platform*, and it should be a valid Android resource name (ie. only lowercase letters, numerals, the underscore, and the period are allowed).

- **iOS** - The preferred way to manage and support images since iOS 9 is to use **Asset Catalog Image Sets**, which should contain all of the versions of an image that are necessary to support various devices and scale factors for an application. For more information, see [Adding Images to an Asset Catalog Image Set](#).
- **Android** - Place images in the `Resources/drawable` directory with **Build Action: AndroidResource**. High- and low-DPI versions of an image can also be supplied (in appropriately named `Resources` subdirectories)

such as **drawable-ldpi**, **drawable-hdpi**, and **drawable-xhdpi**).

- **Universal Windows Platform (UWP)** - Place images in the application's root directory with **Build Action: Content**.

#### IMPORTANT

Prior to iOS 9, images were typically placed in the **Resources** folder with **Build Action: BundleResource**. However, this method of working with images in an iOS app has been deprecated by Apple. For more information, see [Image Sizes and Filenames](#).

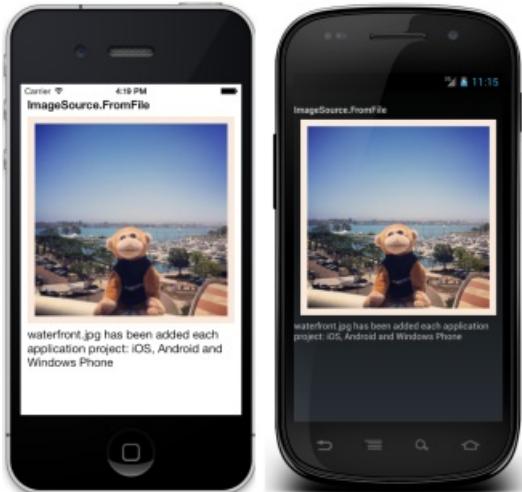
Adhering to these rules for file naming and placement allows the following XAML to load and display the image on all platforms:

```
<Image Source="waterfront.jpg" />
```

The equivalent C# code is as follows:

```
var image = new Image { Source = "waterfront.jpg" };
```

The following screenshots show the result of displaying a local image on each platform:



For more flexibility the `Device.RuntimePlatform` property can be used to select a different image file or path for some or all platforms, as shown in this code example:

```
image.Source = Device.RuntimePlatform == Device.Android ? ImageSource.FromFile("waterfront.jpg") :  
ImageSource.FromFile("Images/waterfront.jpg");
```

#### IMPORTANT

To use the same image filename across all platforms the name must be valid on all platforms. Android drawables have naming restrictions – only lowercase letters, numbers, underscore, and period are allowed – and for cross-platform compatibility this must be followed on all the other platforms too. The example filename **waterfront.png** follows the rules, but examples of invalid filenames include "water front.png", "WaterFront.png", "water-front.png", and "wåterfront.png".

## Native Resolutions (Retina and High-DPI)

iOS, Android, and UWP include support for different image resolutions, where the operating system chooses the appropriate image at runtime based on the device's capabilities. Xamarin.Forms uses the native platforms' APIs for loading local images, so it automatically supports alternate resolutions if the files are correctly named and

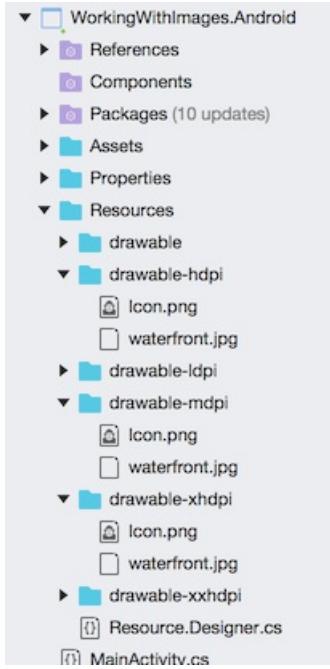
located in the project.

The preferred way to manage images since iOS 9 is to drag images for each resolution required to the appropriate asset catalog image set. For more information, see [Adding Images to an Asset Catalog Image Set](#).

Prior to iOS 9, retina versions of the image could be placed in the **Resources** folder - two and three times the resolution with a **@2x** or **@3x** suffixes on the filename before the file extension (eg. **myimage@2x.png**).

However, this method of working with images in an iOS app has been deprecated by Apple. For more information, see [Image Sizes and Filenames](#).

Android alternate resolution images should be placed in **specially-named directories** in the Android project, as shown in the following screenshot:



UWP image file names [can be suffixed with `.scale-xxx`](#) before the file extension, where `xxx` is the percentage of scaling applied to the asset, e.g. **myimage.scale-200.png**. Images can then be referred to in code or XAML without the scale modifier, e.g. just **myimage.png**. The platform will select the nearest appropriate asset scale based on the display's current DPI.

## Additional Controls that Display Images

Some controls have properties that display an image, such as:

- **Page** - Any page type that derives from **Page** has **Icon** and **BackgroundImage** properties, which can be assigned a local file reference. Under certain circumstances, such as when a **NavigationPage** is displaying a **ContentPage**, the icon will be displayed if supported by the platform.

### IMPORTANT

On iOS, the **Page.Icon** property can't be populated from an image in an asset catalog image set. Instead, load icon images for the **Page.Icon** property from the **Resources** folder in the iOS project.

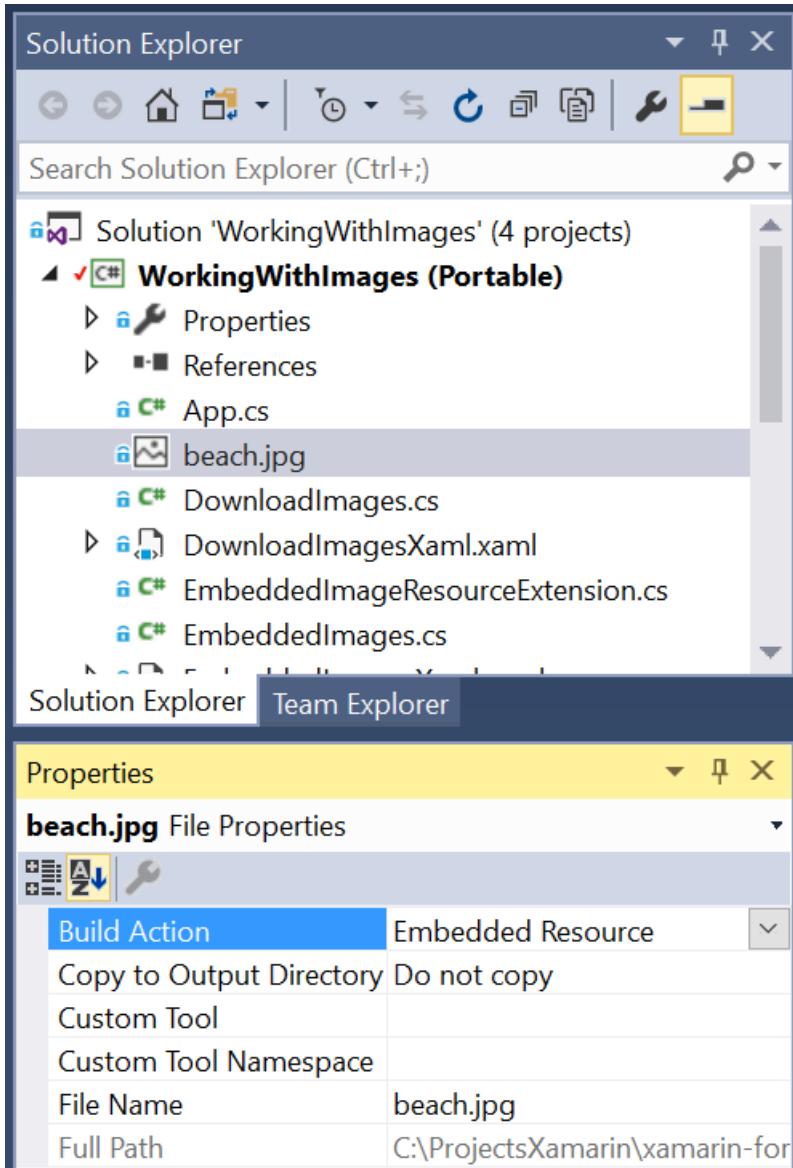
- **ToolbarItem** - Has an **Icon** property that can be set to a local file reference.
- **ImageCell** - Has an **ImageSource** property that can be set to an image retrieved from a local file, an embedded resource, or a URI.

## Embedded Images

Embedded images are also shipped with an application (like local images) but instead of having a copy of the image in each application's file structure the image file is embedded in the assembly as a resource. This method of distributing images is recommended when identical images are used on each platform, and is particularly suited to creating components, as the image is bundled with the code.

To embed an image in a project, right-click to add new items and select the image/s you wish to add. By default the image will have **Build Action: None**; this needs to be set to **Build Action: EmbeddedResource**.

- [Visual Studio](#)
- [Visual Studio for Mac](#)



The **Build Action** can be viewed and changed in the **Properties** window for a file.

In this example the resource ID is **WorkingWithImages.beach.jpg**. The IDE has generated this default by concatenating the **Default Namespace** for this project with the filename, using a period (.) between each value.

If you place embedded images into folders within your project, the folder names are also separated by periods (.) in the resource ID. Moving the **beach.jpg** image into a folder called **MyImages** would result in a resource ID of **WorkingWithImages.MyImages.beach.jpg**

The code to load an embedded image simply passes the **Resource ID** to the `ImageSource.FromResource` method as shown below:

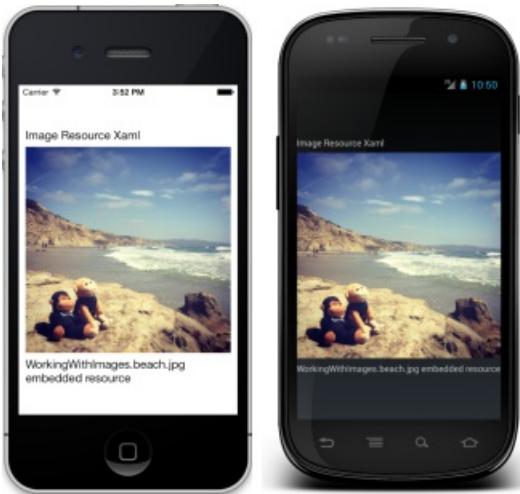
```
var embeddedImage = new Image { Source = ImageSource.FromResource("WorkingWithImages.beach.jpg",  
typeof(EmbeddedImages).GetTypeInfo().Assembly) };
```

#### NOTE

To support displaying embedded images in release mode on the Universal Windows Platform, it's necessary to use the overload of `ImageSource.FromResource` that specifies the source assembly in which to search for the image.

Currently there is no implicit conversion for resource identifiers. Instead, you must use `ImageSource.FromResource` or `new ResourceImageSource()` to load embedded images.

The following screenshots show the result of displaying an embedded image on each platform:



#### Using XAML

Because there is no built-in type converter from `string` to `ResourceImageSource`, these types of images cannot be natively loaded by XAML. Instead, a simple custom XAML markup extension can be written to load images using a **Resource ID** specified in XAML:

```
[ContentProperty (nameof(Source))]  
public class ImageResourceExtension : IMarkupExtension  
{  
    public string Source { get; set; }  
  
    public object ProvideValue(IServiceProvider serviceProvider)  
    {  
        if (Source == null)  
        {  
            return null;  
        }  
  
        // Do your translation lookup here, using whatever method you require  
        var imageSource = ImageSource.FromResource(Source, typeof(ImageResourceExtension).GetTypeInfo().Assembly);  
  
        return imageSource;  
    }  
}
```

#### NOTE

To support displaying embedded images in release mode on the Universal Windows Platform, it's necessary to use the overload of `ImageSource.FromResource` that specifies the source assembly in which to search for the image.

To use this extension add a custom `xmlns` to the XAML, using the correct namespace and assembly values for the project. The image source can then be set using this syntax: `{local:ImageResource WorkingWithImages.beach.jpg}`. A complete XAML example is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:WorkingWithImages;assembly=WorkingWithImages"
    x:Class="WorkingWithImages.EmbeddedImagesXaml">
    <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
        <!-- use a custom Markup Extension -->
        <Image Source="{local:ImageResource WorkingWithImages.beach.jpg}" />
    </StackLayout>
</ContentPage>
```

## Troubleshooting Embedded Images

### Debugging Code

Because it is sometimes difficult to understand why a particular image resource isn't being loaded, the following debug code can be added temporarily to an application to help confirm the resources are correctly configured. It will output all known resources embedded in the given assembly to the Console to help debug resource loading issues.

```
using System.Reflection;
// ...
// NOTE: use for debugging, not in released app code!
var assembly = typeof(EmbeddedImages).GetTypeInfo().Assembly;
foreach (var res in assembly.GetManifestResourceNames())
{
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}
```

### Images Embedded in Other Projects

By default, the `ImageSource.FromResource` method only looks for images in the same assembly as the code calling the `ImageSource.FromResource` method. Using the debug code above you can determine which assemblies contain a specific resource by changing the `typeof()` statement to a `Type` known to be in each assembly.

However, the source assembly being searched for an embedded image can be specified as an argument to the `ImageSource.FromResource` method:

```
var imageSource = ImageSource.FromResource("filename.png", typeof(MyClass).GetTypeInfo().Assembly);
```

## Downloading Images

Images can be automatically downloaded for display, as shown in the following XAML:

```
<?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="WorkingWithImages.DownloadImagesXaml">
    <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
        <Label Text="Image UriSource Xaml" />
        <Image Source="https://xamarin.com/content/images/pages/forms/example-app.png" />
        <Label Text="example-app.png gets downloaded from xamarin.com" />
    </StackLayout>
</ContentPage>
```

The equivalent C# code is as follows:

```
var webImage = new Image { Source = ImageSource.FromUri(new Uri("https://xamarin.com/content/images/pages/forms/example-app.png")) };
```

The `ImageSource.FromUri` method requires a `Uri` object, and returns a new `UriImageSource` that reads from the `Uri`.

There is also an implicit conversion for URI strings, so the following example will also work:

```
webImage.Source = "https://xamarin.com/content/images/pages/forms/example-app.png";
```

The following screenshots show the result of displaying a remote image on each platform:



### Downloaded Image Caching

A `UriImageSource` also supports caching of downloaded images, configured through the following properties:

- `CachingEnabled` - Whether caching is enabled (`true` by default).
- `CacheValidity` - A `TimeSpan` that defines how long the image will be stored locally.

Caching is enabled by default and will store the image locally for 24 hours. To disable caching for a particular image, instantiate the image source as follows:

```
image.Source = new UriImageSource { CachingEnabled = false, Uri="http://server.com/image" };
```

To set a specific cache period (for example, 5 days) instantiate the image source as follows:

```
webImage.Source = new UriImageSource
{
    Uri = new Uri("https://xamarin.com/content/images/pages/forms/example-app.png"),
    CachingEnabled = true,
    CacheValidity = new TimeSpan(5,0,0,0)
};
```

Built-in caching makes it very easy to support scenarios like scrolling lists of images, where you can set (or bind) an image in each cell and let the built-in cache take care of re-loading the image when the cell is scrolled back into view.

## Icons and splash screens

While not related to the [Image](#) view, application icons and splash screens are also an important use of images in Xamarin.Forms projects.

Setting icons and splash screens for Xamarin.Forms apps is done in each of the application projects. This means generating correctly sized images for iOS, Android, and UWP. These images should be named and located according to each platforms' requirements.

## Icons

See the [iOS Working with Images](#), [Google Iconography](#), and [Guidelines for tile and icon assets](#) for more information on creating these application resources.

## Splash screens

Only iOS and UWP applications require a splash screen (also called a startup screen or default image).

Refer to the documentation for [iOS Working with Images](#) and [Splash screens](#) on the Windows Dev Center.

## Summary

Xamarin.Forms offers a number of different ways to include images in a cross-platform application, allowing for the same image to be used across platforms or for platform-specific images to be specified. Downloaded images are also automatically cached, automating a common coding scenario.

Application icon and splash screen images are set-up and configured as for non-Xamarin.Forms applications - follow the same guidance used for platform-specific apps.

## Related Links

- [WorkingWithImages \(sample\)](#)
- [iOS Working with Images](#)
- [Android Iconography](#)
- [Guidelines for tile and icon assets](#)

# Xamarin.Forms ImageButton

11/20/2018 • 6 minutes to read • [Edit Online](#)

The `ImageButton` displays an image and responds to a tap or click that directs an application to carry out a particular task.

The `ImageButton` view combines the `Button` view and `Image` view to create a button whose content is an image. The user presses the `ImageButton` with a finger or clicks it with a mouse to direct the application to carry out a particular task. However, unlike the `Button` view, the `ImageButton` view has no concept of text and text appearance.

## NOTE

While the `Button` view defines an `Image` property, that allows you to display a image on the `Button`, this property is intended to be used when displaying a small icon next to the `Button` text.

The code examples in this guide are taken from the [FormsGallery sample](#).

## Setting the image source

`ImageButton` defines a `Source` property that should be set to the image to display in the button, with the image source being either a file, a URI, a resource, or a stream. For more information about loading images from different sources, see [Images in Xamarin.Forms](#).

The following example shows how to instantiate a `ImageButton` in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FormsGallery.XamlExamples.ImageButtonDemoPage"
    Title="ImageButton Demo">
    <StackLayout>
        <Label Text="ImageButton"
            FontSize="50"
            FontAttributes="Bold"
            HorizontalOptions="Center" />

        <ImageButton Source="XamarinLogo.png"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The `Source` property specifies the image that appears in the `ImageButton`. In this example it's set to a local file that will be loaded from each platform project, resulting in the following screenshots:



By default, the `ImageButton` is rectangular, but you can give it rounded corners by using the `CornerRadius` property. For more information about `ImageButton` appearance, see [ImageButton appearance](#).

The following example shows how to create a page that is functionally equivalent to the previous XAML example, but entirely in C#:

```
public class ImageButtonDemoPage : ContentPage
{
    public ImageButtonDemoPage()
    {
        Label header = new Label
        {
            Text = "ImageButton",
            FontSize = 50,
            FontAttributes = FontAttributes.Bold,
            HorizontalOptions = LayoutOptions.Center
        };

        ImageButton imageButton = new ImageButton
        {
            Source = "XamarinLogo.png",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        // Build the page.
        Title = "ImageButton Demo";
        Content = new StackLayout
        {
            Children = { header, imageButton }
        };
    }
}
```

## Handling ImageButton clicks

`ImageButton` defines a `Clicked` event that is fired when the user taps the `ImageButton` with a finger or mouse pointer. The event is fired when the finger or mouse button is released from the surface of the `ImageButton`. The `ImageButton` must have its `.IsEnabled` property set to `true` to respond to taps.

The following example shows how to instantiate a `ImageButton` in XAML and handle its `Clicked` event:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FormsGallery.XamlExamples.ImageButtonDemoPage"
    Title="ImageButton Demo">
    <StackLayout>
        <Label Text="ImageButton"
            FontSize="50"
            FontAttributes="Bold"
            HorizontalOptions="Center" />

        <ImageButton Source="XamarinLogo.png"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Clicked="OnImageButtonClicked" />

        <Label x:Name="label"
            Text="0 ImageButton clicks"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

The `clicked` event is set to an event handler named `OnImageButtonClicked` that is located in the code-behind file:

```

public partial class ImageButtonDemoPage : ContentPage
{
    int clickTotal;

    public ImageButtonDemoPage()
    {
        InitializeComponent();
    }

    void OnImageButtonClicked(object sender, EventArgs e)
    {
        clickTotal += 1;
        label.Text = $"{clickTotal} ImageButton click{(clickTotal == 1 ? "" : "s")}";
    }
}

```

When the `ImageButton` is tapped, the `OnImageButtonClicked` method executes. The `sender` argument is the `ImageButton` responsible for this event. You can use this to access the `ImageButton` object, or to distinguish between multiple `ImageButton` objects sharing the same `Clicked` event.

This particular `clicked` handler increments a counter and displays the counter value in a `Label`:



The following example shows how to create a page that is functionally equivalent to the previous XAML example, but entirely in C#:

```

public class ImageButtonDemoPage : ContentPage
{
    Label label;
    int clickTotal = 0;

    public ImageButtonDemoPage()
    {
        Label header = new Label
        {
            Text = "ImageButton",
            FontSize = 50,
            FontAttributes = FontAttributes.Bold,
            HorizontalOptions = LayoutOptions.Center
        };

        ImageButton(imageButton = new ImageButton
        {
            Source = "XamarinLogo.png",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        });
        imageButton.Clicked += OnImageButtonClicked;

        label = new Label
        {
            Text = "0 ImageButton clicks",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        // Build the page.
        Title = "ImageButton Demo";
        Content = new StackLayout
        {
            Children =
            {
                header,
                imageButton,
                label
            }
        };
    }

    void OnImageButtonClicked(object sender, EventArgs e)
    {
        clickTotal += 1;
        label.Text = $"{clickTotal} ImageButton click{(clickTotal == 1 ? "" : "s")}";
    }
}

```

## Disabling the ImageButton

Sometimes an application is in a particular state where a particular `ImageButton` click is not a valid operation. In those cases, the `ImageButton` should be disabled by setting its `IsEnabled` property to `false`.

## Using the command interface

It is possible for an application to respond to `ImageButton` taps without handling the `Clicked` event. The `ImageButton` implements an alternative notification mechanism called the *command* or *commanding* interface. This consists of two properties:

- `Command` of type `ICommand`, an interface defined in the `System.Windows.Input` namespace.

- `CommandParameter` property of type `Object`.

This approach is suitable in connection with data-binding, and particularly when implementing the Model-View-ViewModel (MVVM) architecture.

For more information about using the command interface, see [Using the command interface](#) in the [Button](#) guide.

## Pressing and releasing the ImageButton

Besides the `Clicked` event, `ImageButton` also defines `Pressed` and `Released` events. The `Pressed` event occurs when a finger presses on a `ImageButton`, or a mouse button is pressed with the pointer positioned over the `ImageButton`. The `Released` event occurs when the finger or mouse button is released. Generally, the `Clicked` event is also fired at the same time as the `Released` event, but if the finger or mouse pointer slides away from the surface of the `ImageButton` before being released, the `Clicked` event might not occur.

For more information about these events, see [Pressing and releasing the button](#) in the [Button](#) guide.

## ImageButton appearance

In addition to the properties that `ImageButton` inherits from the `View` class, `ImageButton` also defines several properties that affect its appearance:

- `Aspect` is how the image will be scaled to fit the display area.
- `BorderColor` is the color of an area surrounding the `ImageButton`.
- `BorderWidth` is the width of the border.
- `CornerRadius` is the corner radius of the `ImageButton`.

The `Aspect` property can be set to one of the members of the `Aspect` enumeration:

- `Fill` - stretches the image to completely and exactly fill the `ImageButton`. This may result in the image being distorted.
- `AspectFill` - clips the image so that it fills the `ImageButton` while preserving the aspect ratio.
- `AspectFit` - letterboxes the image (if necessary) so that the entire image fits into the `ImageButton`, with blank space added to the top/bottom or sides depending on whether the image is wide or tall. This is the default value of the `Aspect` enumeration.

### NOTE

The `ImageButton` class also has `Margin` and `Padding` properties that control the layout behavior of the `ImageButton`. For more information, see [Margin and Padding](#).

## ImageButton visual states

`ImageButton` has a `Pressed` `VisualState` that can be used to initiate a visual change to the `ImageButton` when pressed by the user, provided that it's enabled.

The following XAML example shows how to define a visual state for the `Pressed` state:

```
<ImageButton Source="XamarinLogo.png"
    ...
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="Scale"
                        Value="1" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Pressed">
                <VisualState.Setters>
                    <Setter Property="Scale"
                        Value="0.8" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</ImageButton>
```

The `Pressed` `VisualState` specifies that when the `ImageButton` is pressed, its `Scale` property will be changed from its default value of 1 to 0.8. The `Normal` `VisualState` specifies that when the `ImageButton` is in a normal state, its `Scale` property will be set to 1. Therefore, the overall effect is that when the `ImageButton` is pressed, it's rescaled to be slightly smaller, and when the `ImageButton` is released, it's rescaled to its default size.

For more information about visual states, see [The Xamarin.Forms Visual State Manager](#).

## Related links

- [FormsGallery sample](#)

# Layouts in Xamarin.Forms

7/12/2018 • 8 minutes to read • [Edit Online](#)

Xamarin.Forms has several layouts and features for organizing content on screen.

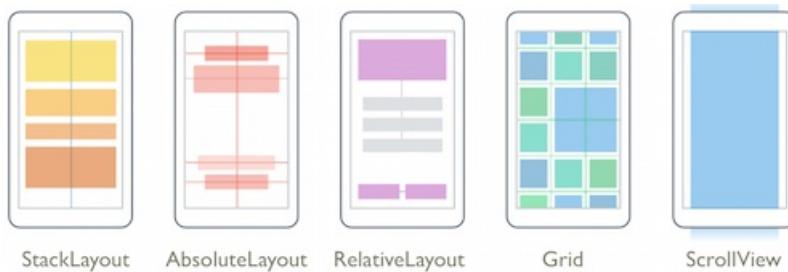
## Xamarin.Forms Layouts, by [Xamarin University](#)

Each layout control is described below, as well as details on how to handle screen orientation changes:

- **StackLayout** – used to arrange views linearly, either horizontally or vertically. Views in a StackLayout can be aligned to the center, left or right of the layout.
- **AbsoluteLayout** – used to arrange views by setting coordinates & size in terms of absolute values or ratios. AbsoluteLayout can be used to layer views as well as anchor them to the left, right or center.
- **RelativeLayout** – used to arrange views by setting constraints relative to their parent's dimensions & position.
- **Grid** – used to arrange views in a grid. Rows and columns can be specified in terms of absolute values or ratios.
- **FlexLayout** – used to arrange views horizontally or vertically with wrapping.
- **ScrollView** – used to provide scrolling when a view can't fit entirely within the bounds of the screen.
- **LayoutOptions** – define alignment and expansion for a view, relative to its parent.
- **Input Transparency** – specifies whether an element receives input.
- **Margin and Padding** – demonstrates how to control layout behavior when an element is rendered in the user interface.
- **Device Orientation** – explains how to handle device orientation changes.
- **Layout on tablet and desktop devices** – shows how to optimize for larger screens on each platform.
- **Creating a Custom Layout** – explains how to create a custom layout class.
- **Layout Compression** – removes specified layout from the visual tree in an attempt to improve page rendering performance.

Platform controls can also be used directly in Xamarin.Forms layouts with **Native Embedding** (new in Xamarin.Forms 2.2), and you can [create custom layouts](#) to meet specific requirements.

The following graphic visualizes the layout controls:



## Choosing the Right Layout

The layouts you choose in your app can either help or hurt you as you're creating an attractive and usable Xamarin.Forms app. Taking some time to consider how each layout works can help you write cleaner and more scalable UI code. A screen can have a combination of different layouts to achieve a specific design.

### StackLayout

The `StackLayout` is used for displaying views along a line that is either horizontal or vertical. Position and size within the layout is determined based on a view's `HeightRequest`, `WidthRequest`, `HorizontalOptions` and

`VerticalOptions` . `StackLayout` is often used as the base layout, arranging other layouts on the screen.

For an example of when `StackLayout` would be a good choice, consider an app that needs to display a button and a label, with the label left-aligned and the button right-aligned.

```
<StackLayout Orientation="Horizontal">
    <Label HorizontalOptions="StartAndExpand" Text="Label" />
    <Button HorizontalOptions="End" Text="Button" />
</StackLayout>
```

## FlexLayout

The `FlexLayout` is similar to `StackLayout` in that it displays child views either horizontally or vertically:

```
<FlexLayout Direction="Column"
            AlignItems="Center"
            JustifyContent="SpaceEvenly">

    <Label Text="FlexLayout in Action" />
    <Button Text="Button" />
    <Label Text="Another Label" />
</FlexLayout>
```

However, if there are too many children to fit in a single row or column, `FlexLayout` is also capable of wrapping those views. `FlexLayout` is based on the CSS Flexible Box Layout Module, and has many of the same built-in options for positioning and aligning its children.

## AbsoluteLayout

The `AbsoluteLayout` is used for displaying views, with size and position being specified either as explicit values or values relative to the size of the layout. Unlike `StackLayout` and `Grid`, `AbsoluteLayout` allows child views to overlap. Unlike `RelativeLayout`, `AbsoluteLayout` doesn't let you place elements off screen.

For an example of when `AbsoluteLayout` would be a good choice, consider an app that needs to present collections of objects as stacks. This is often seen when presenting albums of photos or songs. The following code gives the appearance of a pile, with elements rotated to hint at the contents of the pile:

In XAML:

```
<AbsoluteLayout Padding="15">
    <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
Rotation="30"
    Source="bottom.png" />
    <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
Rotation="60"
    Source="middle.png" />
    <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
Source="cover.png" />
</AbsoluteLayout>
```

Note the following aspects of the above code:

- Each `Image` is displayed in the same position (in the middle of the horizontal space)
- The `Padding` is considered by `AbsoluteLayout`, unlike `RelativeLayout`, which ignores it.
- `AbsoluteLayout.LayoutFlags` specifies how the layout bounds will be interpreted. In this case `PositionProportional`, means that the coordinates will be a ratio of the size of the layout, while the size will be interpreted as an explicit size.
- `AbsoluteLayout.Layoutbounds` specifies the horizontal position, vertical position, width and height in that order.

## RelativeLayout

The `RelativeLayout` is used for displaying views, with size and position specified as values relative to the values of the layout or another view. Relative values do not need to match the corresponding value on the related view. As an example, it is possible to set a view's `Width` property to be proportional to another view's `x` property.

RelativeLayout can be used to create UIs that scale proportionally across device sizes. The following XAML implements a design with boxes at the topmost corners, with a flagpole with flag in the center:

```
<RelativeLayout HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
    <BoxView Color="Blue" HeightRequest="50" WidthRequest="50"
        RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = 0}"
        RelativeLayout.YConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Height, Factor = 0}" />
    <BoxView Color="Red" HeightRequest="50" WidthRequest="50"
        RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = .9}"
        RelativeLayout.YConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Height, Factor = 0}" />
    <BoxView Color="Gray" WidthRequest="15" x:Name="pole"
        RelativeLayout.HeightConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Height,
        Factor=.75}"
        RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = .45}"
        RelativeLayout.YConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Height, Factor = .25}" />
    <BoxView Color="Green"
        RelativeLayout.HeightConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Height, Factor=.10,
        Constant=10}"
        RelativeLayout.WidthConstraint=" {ConstraintExpression Type=RelativeToParent, Property=Width, Factor=.2,
        Constant=20}"
        RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToView, ElementName=pole, Property=X,
        Constant=15}"
        RelativeLayout.YConstraint=" {ConstraintExpression Type=RelativeToView, ElementName=pole, Property=Y,
        Constant=0}" />
</RelativeLayout>
```

Note the following aspects of the above code:

- Both positions and sizes are specified as constraints.
- The flagpole is named so that the flag's (green box's) position can be set relative to the flagpole.
- The constraint expressions have `Factor` and `Constant` properties, which can be used to define positions and sizes as multiples (or fractions) of properties of other objects, plus a constant. Constants can be negative.

## Grid

The `Grid` is used for displaying elements in rows & columns. Note that the Grid is not a table, so it does not have the concept of cells, header & footer rows, or borders between rows & columns. In general, Grid is not appropriate for displaying tabular data. For that use, consider a [ListView](#) or [TableView](#).

For an example of when a `Grid` is the right layout to use, consider a numeric input for a calculator. A numeric input for a calculator might consist of four rows and three columns, each with a button. The following code implements this design:

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Button Text="1" Grid.Row="0" Grid.Column="0" />
    <Button Text="2" Grid.Row="0" Grid.Column="1" />
    <Button Text="3" Grid.Row="0" Grid.Column="2" />
    <Button Text="4" Grid.Row="1" Grid.Column="0" />
    <Button Text="5" Grid.Row="1" Grid.Column="1" />
    <Button Text="6" Grid.Row="1" Grid.Column="2" />
    <Button Text="7" Grid.Row="2" Grid.Column="0" />
    <Button Text="8" Grid.Row="2" Grid.Column="1" />
    <Button Text="9" Grid.Row="2" Grid.Column="2" />
    <Button Text="0" Grid.Row="3" Grid.Column="1" />
    <Button Text="<-" Grid.Row="3" Grid.Column="2" />
</Grid>

```

Note the following aspects of the above code:

- Grids and Columns are explicitly specified, not inferred from the content.
- `Height` and `Width` values can be set to star, which means that the Grid will set those values to fill the available space.
- Each button's position is specified by `Grid.Row` & `Grid.Column` properties.

## LayoutOptions

The `LayoutOptions` structure can be used to define alignment and expansion for a view, relative to its parent.

## Margin and Padding

The `Margin` and `Padding` properties control layout behavior when an element is rendered in the user interface.

## Input Transparency

Each element has an `InputTransparent` property that's used to define whether the element receives input. Its default value is `false`, ensuring that the element receives input.

When this property is set on a container class, such as a layout class, its value transfers to child elements.

Therefore, setting the `InputTransparent` property to `true` on a layout class will result in the elements within the layout not receiving input.

## Device Orientation

Xamarin.Forms and its built-in layouts are capable of handling changes in device orientation. Consider which orientations your app will support, as well as how you'll make use of the space provided in landscape and portrait modes.

## Layout for Tablet and Desktop apps

iOS, Android, and Universal Windows Platform all support larger screen sizes on tablet devices (as well as laptops and desktops for Windows). Xamarin.Forms lets you optimize your app for larger screens by detecting the device type and either adjusting the page layout, or using a totally different page altogether for larger screens.

## Creating a Custom Layout

Xamarin.Forms defines four layout classes - `StackLayout`, `AbsoluteLayout`, `RelativeLayout`, and `Grid`, and each

arranges its children in a different way. However, sometimes it's necessary to organize page content using a layout not provided by Xamarin.Forms. This article explains how to write a custom layout class, and demonstrates an orientation-sensitive `WrapLayout` class that arranges its children horizontally across the page, and then wraps the display of subsequent children to additional rows.

## Layout Compression

Layout compression removes specified layouts from the visual tree in an attempt to improve page rendering performance. The performance benefit that this delivers varies depending on the complexity of a page, the version of the operating system being used, and the device on which the application is running. However, the biggest performance gains will be seen on older devices.

## Making Your Choice

Be aware that in most cases, more than one layout choice can be used to implement your desired design. When there are multiple valid choices, consider which approach will be the easiest for your situation. Most designs can't be realized with just one layout, so nest layouts as needed to create more complex designs.

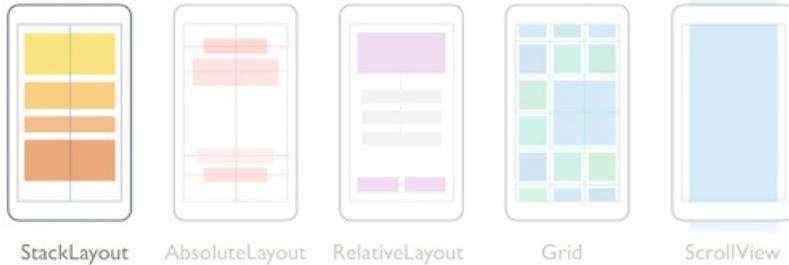
## Related Links

- [Apple's Human Interface Guidelines](#)
- [Android Design Website](#)
- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# Xamarin.Forms StackLayout

6/8/2018 • 4 minutes to read • [Edit Online](#)

`StackLayout` organizes views in a one-dimensional line ("stack"), either horizontally or vertically. Views in a `StackLayout` can be sized based on the space in the layout using layout options. Positioning is determined by the order views were added to the layout and the layout options of the views.



## Purpose

`StackLayout` is less complex than other views. Simple linear interfaces can be created by just adding views to a `StackLayout`, and more complex interfaces created by nesting them.

## Usage & Behavior

### Spacing

By default, `StackLayout` will add a 6px margin between views. This can be controlled or set to have no margin by setting the `Spacing` property on `StackLayout`. The following demonstrates how to set spacing and the effect of different spacing options:

In XAML:

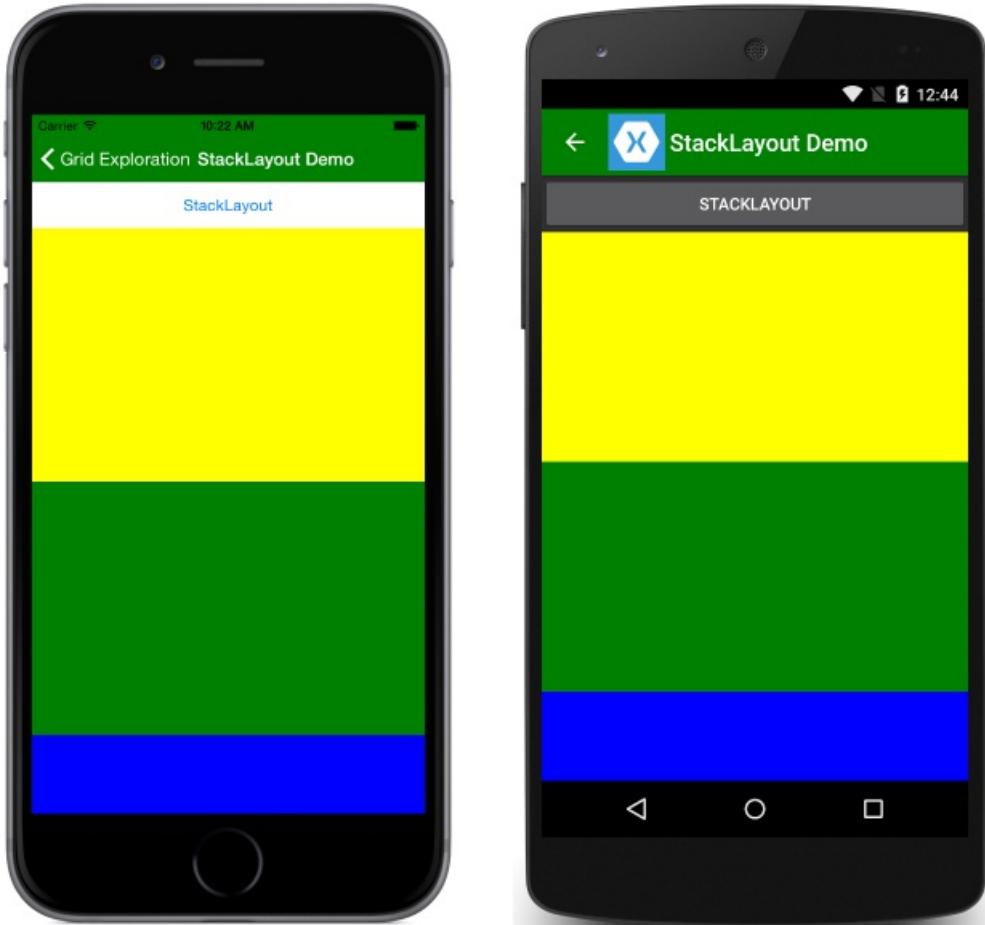
```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="LayoutSamples.StackLayoutDemo"
  Title="StackLayout Demo">
  <ContentPage.Content>
    <StackLayout Spacing="10" x:Name="layout">
      <Button Text="StackLayout" VerticalOptions="Start"
        HorizontalOptions="FillAndExpand" />
      <BoxView Color="Yellow" VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
      <BoxView Color="Green" VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
      <BoxView HeightRequest="75" Color="Blue" VerticalOptions="End"
        HorizontalOptions="FillAndExpand" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

In C#:

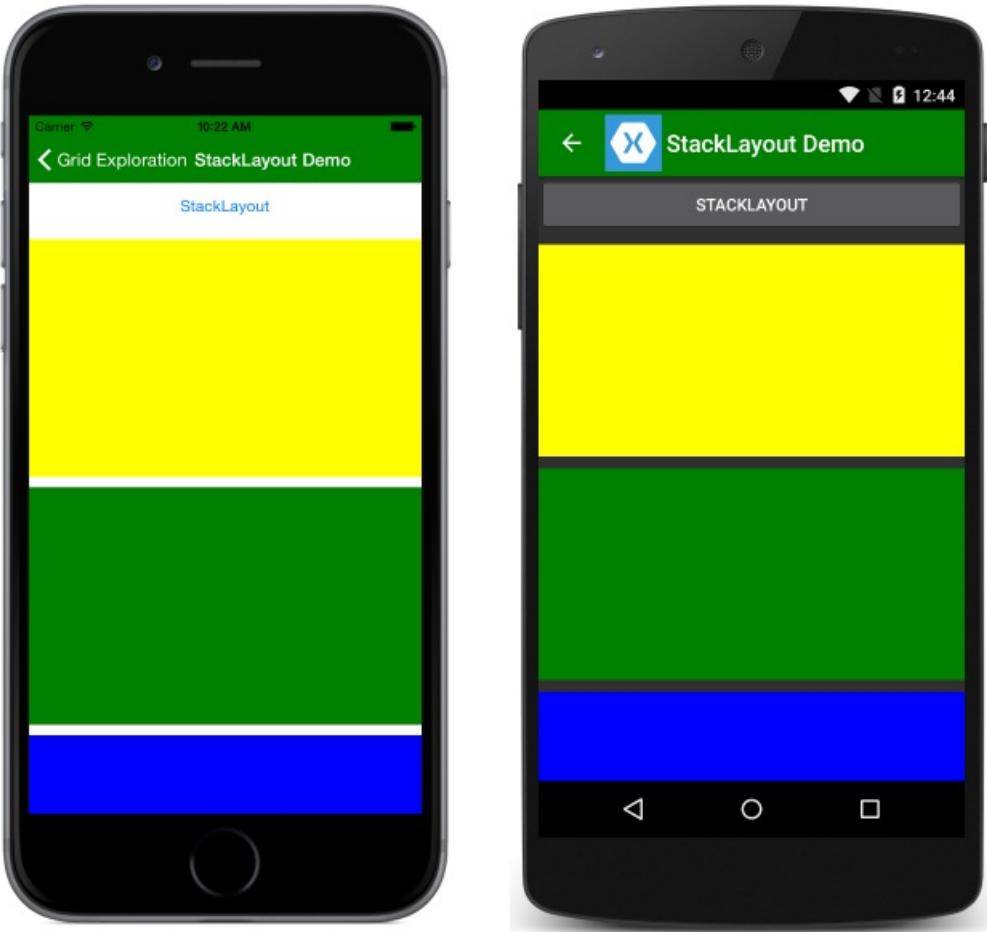
```
public class StackLayoutCode : ContentPage
{
    public StackLayoutCode ()
    {
        var layout = new StackLayout ();
        var button = new Button { Text = "StackLayout", VerticalOptions = LayoutOptions.Start,
            HorizontalOptions = LayoutOptions.FillAndExpand };
        var yellowBox = new BoxView { Color = Color.Yellow, VerticalOptions = LayoutOptions.FillAndExpand,
HorizontalOptions = LayoutOptions.FillAndExpand };
        var greenBox = new BoxView { Color = Color.Green, VerticalOptions = LayoutOptions.FillAndExpand,
HorizontalOptions = LayoutOptions.FillAndExpand };
        var blueBox = new BoxView { Color = Color.Blue, VerticalOptions = LayoutOptions.FillAndExpand,
            HorizontalOptions = LayoutOptions.FillAndExpand, HeightRequest = 75 };

        layout.Children.Add(button);
        layout.Children.Add(yellowBox);
        layout.Children.Add(greenBox);
        layout.Children.Add(blueBox);
        layout.Spacing = 10;
        Content = layout;
    }
}
```

Spacing = 0:



Spacing of ten:



## Sizing

The size of a view in a `StackLayout` depends on both the height and width requests and the layout options.

`StackLayout` will enforce padding. The following `LayoutOption`s will cause views to take up as much space as is available from the layout:

- **CenterAndExpand** – centers the view within the layout and expands to take up as much space as the layout will give it.
- **EndAndExpand** – places the view at the end of the layout (bottom or right-most boundary) and expands to take up as much space as the layout will give it.
- **FillAndExpand** – places the view so that it has no padding and takes up as much space as the layout will give it.
- **StartAndExpand** – places the view at the start of the layout and takes up as much space as the parent will give.

For more information, see [Expansion](#).

## Positioning

Views in a `StackLayout` can be positioned and sized using `LayoutOptions`. Each view can be given

`VerticalOptions` and `HorizontalOptions`, defining how the views will position themselves relative to the layout.

The following predefined `LayoutOptions` are available:

- **Center** – centers the view within the layout.
- **End** – places the view at the end of the layout (bottom or right-most boundary).
- **Fill** – places the view so that it has no padding.
- **Start** – places the view at the start of the layout.

The following code demonstrates setting layout options:

In XAML:

```
<?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="LayoutSamples.StackLayoutDemo"
  Title="StackLayout Demo">
  <ContentPage.Content>
    <StackLayout x:Name="layout">
      <Button VerticalOptions="Start"
        HorizontalOptions="FillAndExpand" />
      <BoxView VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
      <BoxView VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
      <BoxView HeightRequest="75" VerticalOptions="End"
        HorizontalOptions="FillAndExpand" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

In C#:

```
public class StackLayoutCode : ContentPage
{
  public StackLayoutCode ()
  {
    var layout = new StackLayout ();
    var button = new Button { VerticalOptions = LayoutOptions.Start,
      HorizontalOptions = LayoutOptions.FillAndExpand };
    var oneBox = new BoxView { VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
      LayoutOptions.FillAndExpand };
    var twoBox = new BoxView { VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
      LayoutOptions.FillAndExpand };
    var threeBox = new BoxView { VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
      LayoutOptions.FillAndExpand };

    layout.Children.Add(button);
    layout.Children.Add(oneBox);
    layout.Children.Add(twoBox);
    layout.Children.Add(threeBox);
    Content = layout;
  }
}
```

For more information, see [Alignment](#).

## Exploring a Complex Layout

Each of the layouts have strengths and weaknesses for creating particular layouts. Throughout this series of layout articles, a sample app has been created with the same page layout implemented using three different layouts.

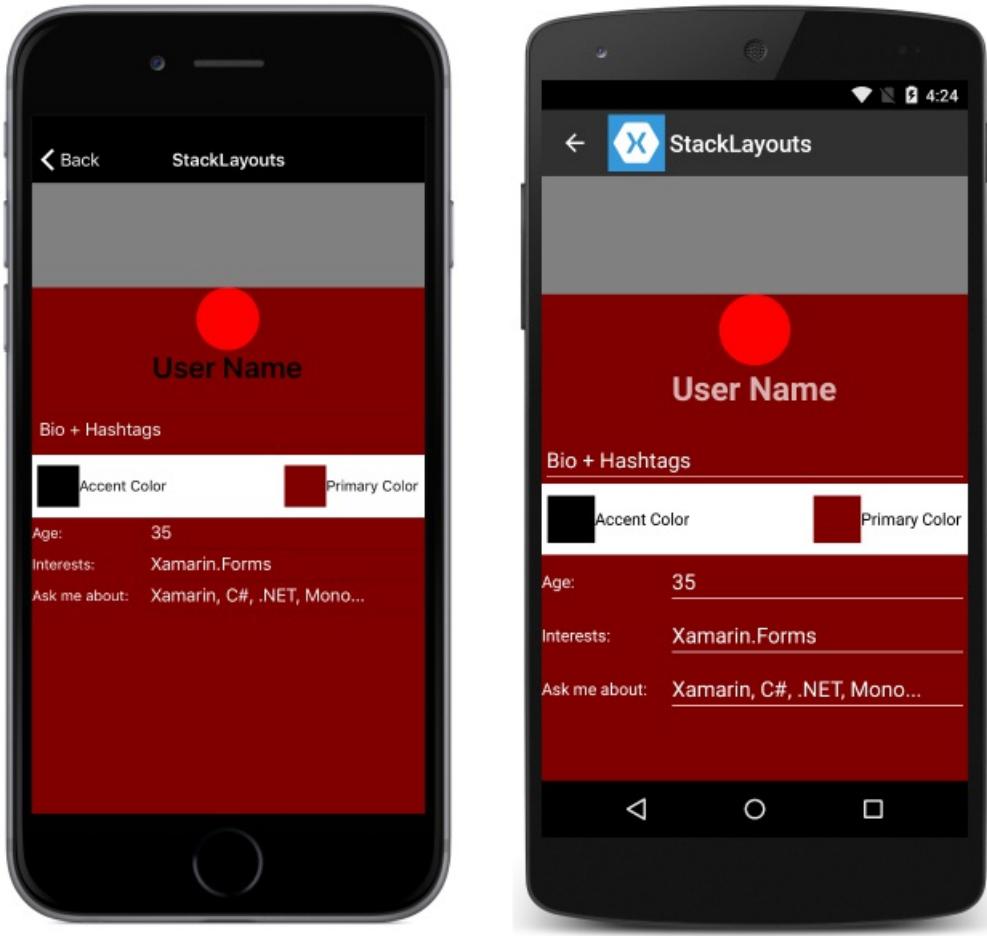
Consider the following XAML:

```

<?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="TheBusinessTumble.StackLayoutPage"
  BackgroundColor="Maroon"
  Title="StackLayouts">
  <ContentPage.Content>
    <ScrollView>
      <StackLayout Spacing="0" Padding="0" BackgroundColor="Maroon">
        <BoxView HorizontalOptions="FillAndExpand" HeightRequest="100"
          VerticalOptions="Start" Color="Gray" />
        <Button BorderRadius="30" HeightRequest="60" WidthRequest="60"
          BackgroundColor="Red" HorizontalOptions="Center" VerticalOptions="Start" />
        <StackLayout HeightRequest="100" VerticalOptions="Start" HorizontalOptions="FillAndExpand"
          Spacing="20" BackgroundColor="Maroon">
          <Label Text="User Name" FontSize="28" HorizontalOptions="Center"
            VerticalOptions="Center" FontAttributes="Bold" />
          <Entry Text="Bio + Hashtags" TextColor="White"
            BackgroundColor="Maroon" HorizontalOptions="FillAndExpand" VerticalOptions="CenterAndExpand" />
        </StackLayout>
        <StackLayout Orientation="Horizontal" HeightRequest="50" BackgroundColor="White" Padding="5">
          <StackLayout Spacing="0" BackgroundColor="White" Orientation="Horizontal" HorizontalOptions="Start">
            <BoxView BackgroundColor="Black" WidthRequest="40" HeightRequest="40"
              HorizontalOptions="StartAndExpand" VerticalOptions="Center" />
            <Label FontSize="14" TextColor="Black" Text="Accent Color" HorizontalOptions="StartAndExpand"
              VerticalOptions="Center" />
          </StackLayout>
          <StackLayout Spacing="0" BackgroundColor="White" Orientation="Horizontal"
            HorizontalOptions="EndAndExpand">
            <BoxView BackgroundColor="Maroon" WidthRequest="40" HeightRequest="40" HorizontalOptions="Start"
              VerticalOptions="Center" />
            <Label FontSize="14" TextColor="Black" Text="Primary Color" HorizontalOptions="StartAndExpand"
              VerticalOptions="Center" />
          </StackLayout>
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Age:" TextColor="White" HorizontalOptions="Start"
            VerticalOptions="Center" WidthRequest="100" />
          <Entry HorizontalOptions="FillAndExpand" Text="35" TextColor="White" BackgroundColor="Maroon" />
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Interests:" TextColor="White"
            HorizontalOptions="Start" VerticalOptions="Center" WidthRequest="100" />
          <Entry HorizontalOptions="FillAndExpand" Text="Xamarin.Forms" TextColor="White"
            BackgroundColor="Maroon" />
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Ask me about:" TextColor="White"
            HorizontalOptions="Start" VerticalOptions="Center" WidthRequest="100" />
          <Entry HorizontalOptions="FillAndExpand" Text="Xamarin, C#, .NET, Mono..." TextColor="White"
            BackgroundColor="Maroon" />
        </StackLayout>
      </ScrollView>
    </ContentPage.Content>
  </ContentPage>

```

The above code results in the following layout:



Notice that `StackLayout`s are nested, because in some cases nesting layouts can be easier than presenting all elements within the same layout. Also notice that, because `StackLayout` doesn't support overlapping items, the page doesn't have some of the layout niceties found in the pages for the other layouts.

## Related Links

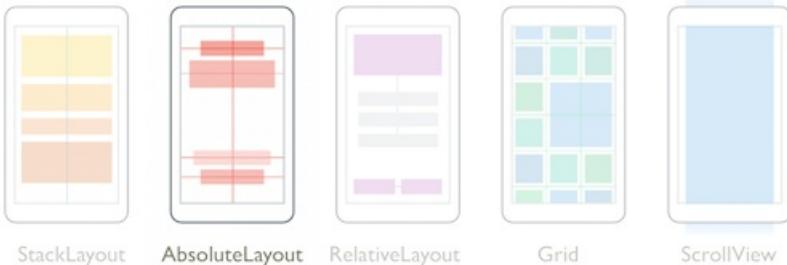
- [LayoutOptions](#)
- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# Xamarin.Forms AbsoluteLayout

7/12/2018 • 7 minutes to read • [Edit Online](#)

`AbsoluteLayout` positions and sizes child elements proportional to its own size and position or by absolute values.

Child views may be positioned and sized using proportional values or static values, and proportional and static values can be mixed.



This article will cover:

- **Purpose** – common uses for `AbsoluteLayout`.
- **Usage** – how to use `AbsoluteLayout` to achieve your desired design.
  - **Proportional Layouts** – understand how proportional values work in an `AbsoluteLayout`.
  - **Specifying Values** – understand how proportional and absolute values are specified.
  - **Proportional Values** – understand how proportional values work.
  - **Absolute Values** – understand how absolute values work.

## Purpose

Because of the positioning model of `AbsoluteLayout`, the layout makes it relatively straightforward to position elements so that they are flush with any side of the layout, or centered. With proportional sizes and positions, elements in an `AbsoluteLayout` can scale automatically to any view size. For items where only the position but not the size should scale, absolute and proportional values can be mixed.

`AbsoluteLayout` could be used anywhere elements need to be positioned within a view and is especially useful when aligning elements to edges.

## Usage

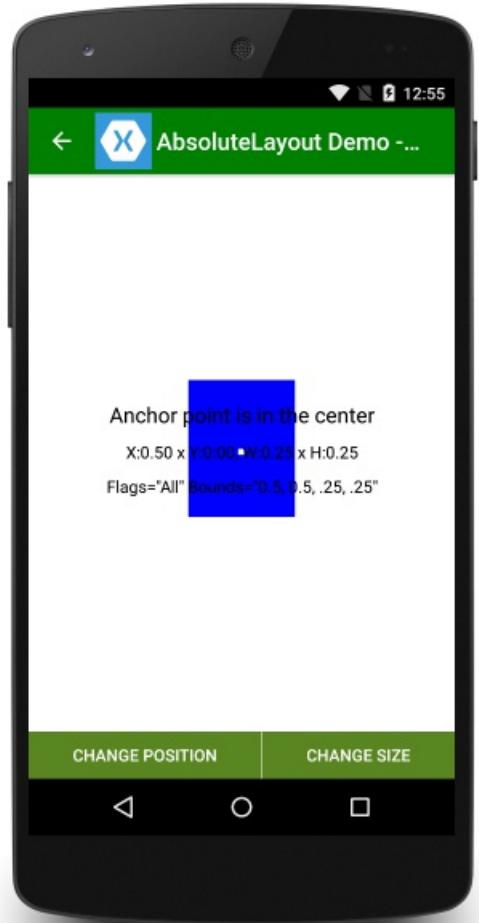
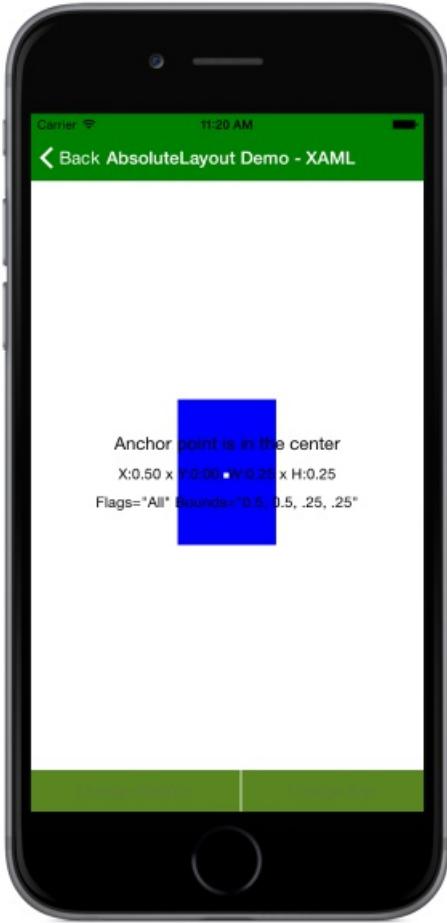
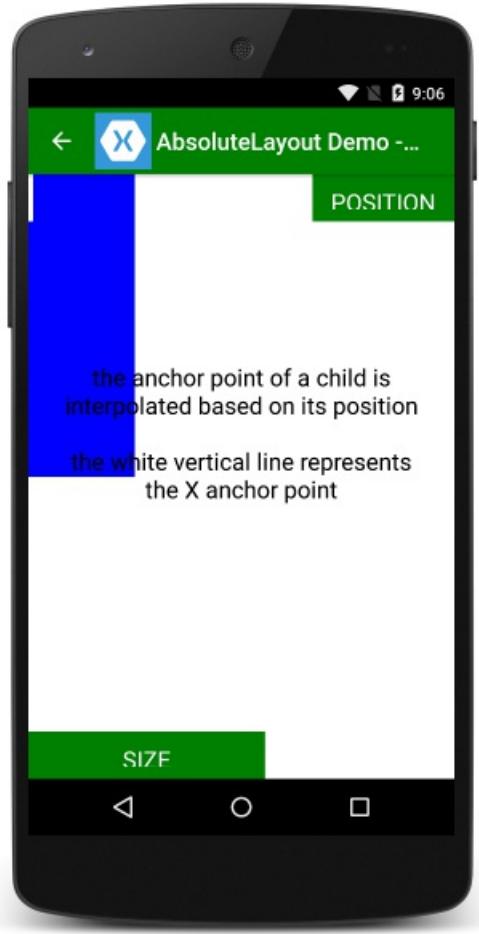
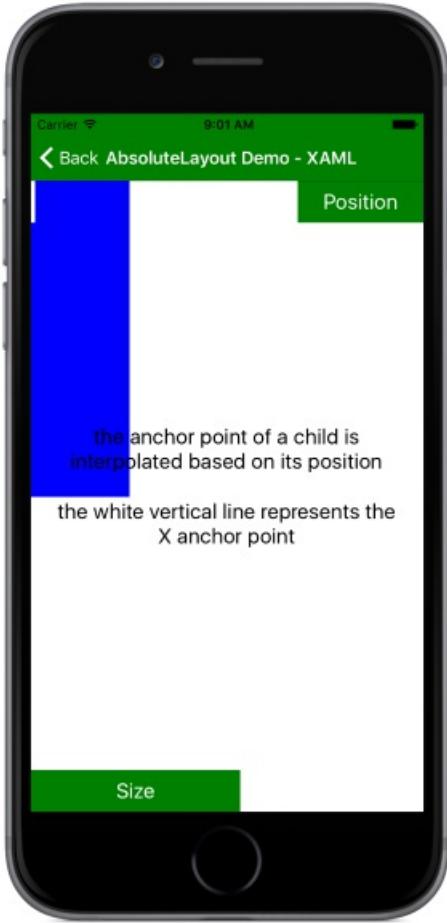
### Proportional Layouts

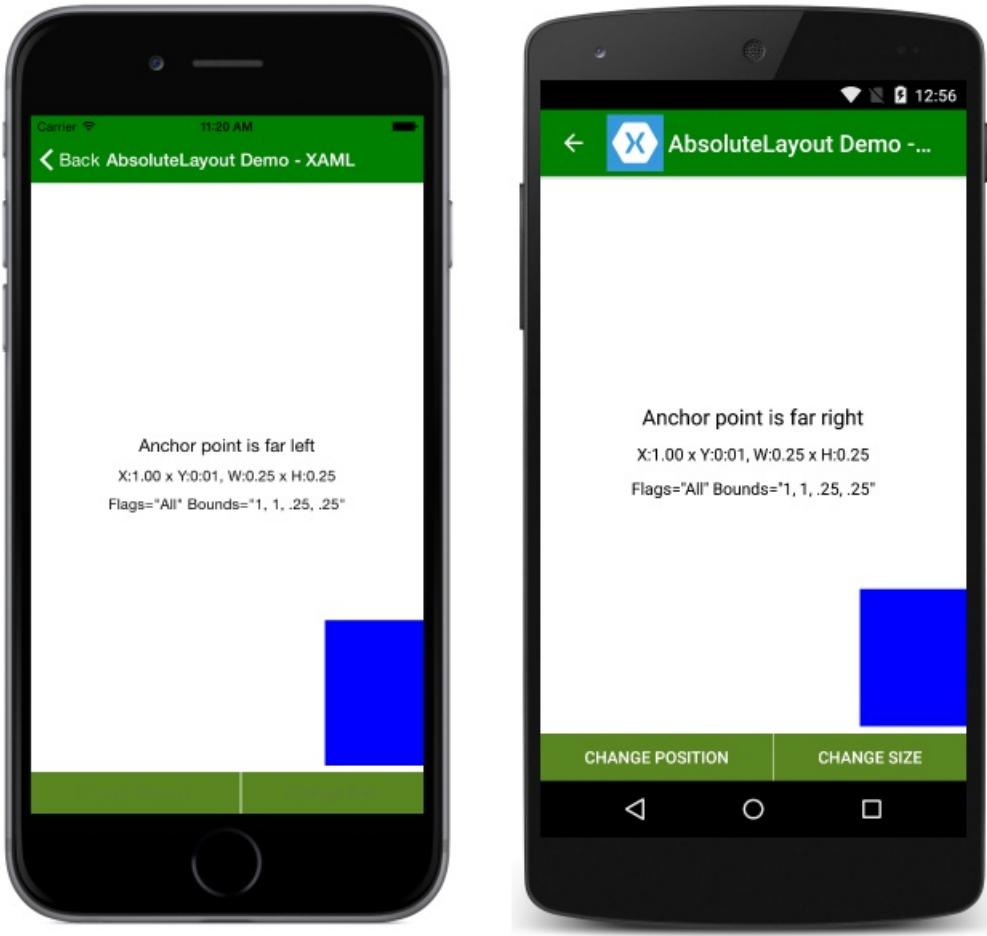
`AbsoluteLayout` has a unique anchor model whereby the anchor of the element is positioned relative to its element as the element is positioned relative to the layout when proportional positioning is used. When absolute positioning is used, the anchor is at (0,0) within the view. This has two important consequences:

- Elements cannot be positioned off screen using proportional values.
- Elements can be reliably positioned along any side of the layout or in the center, regardless of the size of the layout or device.

`AbsoluteLayout`, like `RelativeLayout`, is able to position elements so that they overlap.

Note in the following screenshot, the anchor of the box is a white dot. Notice the relationship between the anchor and the box as it moves through the layout:





## Specifying Values

Views within an `AbsoluteLayout` are positioned using four values:

- **X** – the x (horizontal) position of the view's anchor
- **Y** – the y (vertical) position of the view's anchor
- **Width** – the width of the view
- **Height** – the height of the view

Each of those values can be set as a [proportional](#) value or an [absolute](#) value.

Values are specified as a combination of bounds and a flag. `LayoutBounds` is a `Rectangle` consisting of four values: `x`, `y`, `width`, `height`.

## AbsoluteLayoutFlags

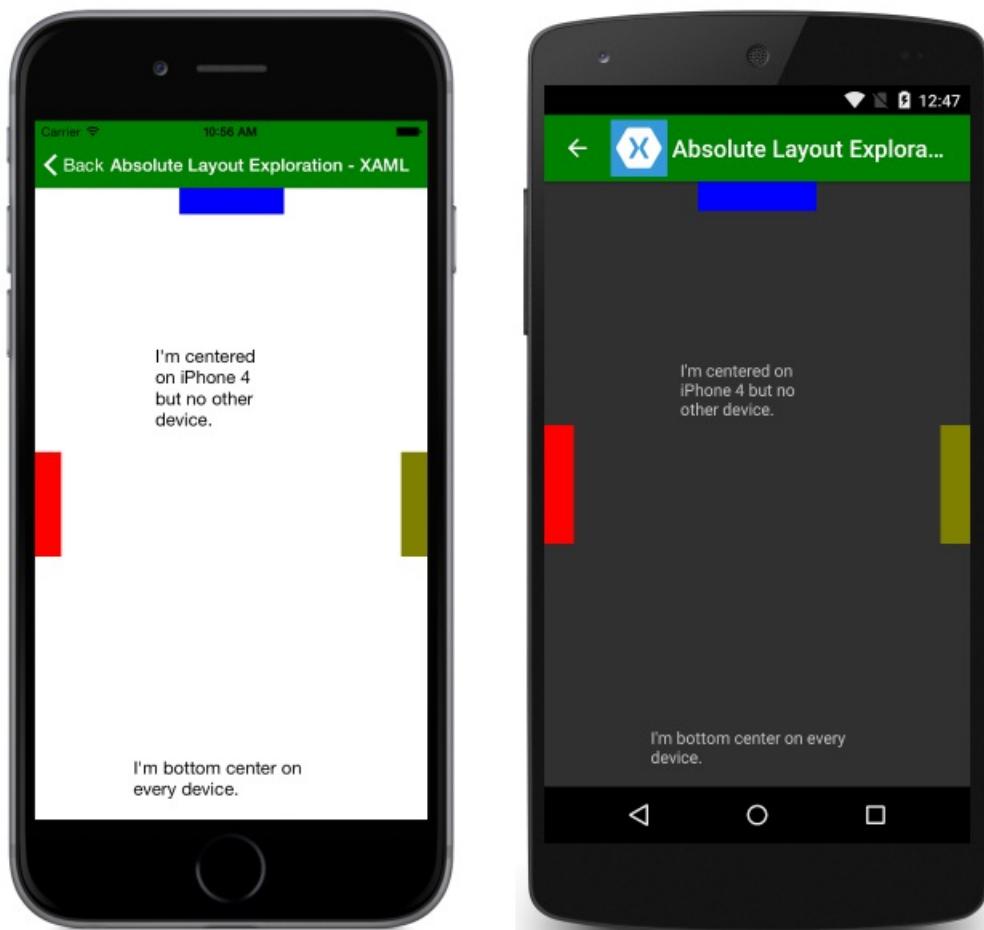
`AbsoluteLayoutFlags` specifies how values will be interpreted and has the following predefined options:

- **None** – interprets all values as absolute. This is the default value if no layout flags are specified.
- **All** – interprets all values as proportional.
- **WidthProportional** – interprets the `Width` value as proportional and all other values as absolute.
- **HeightProportional** – interprets only the height value as proportional with all other values absolute.
- **XProportional** – interprets the `x` value as proportional, while treating all other values as absolute.
- **YProportional** – interprets the `y` value as proportional, while treating all other values as absolute.
- **PositionProportional** – interprets the `x` and `y` values as proportional, while the size values are interpreted as absolute.
- **SizeProportional** – interprets the `Width` and `Height` values as proportional while the position values are absolute.

In XAML, bounds and flags are set as part of the definition of views within the layout, using the

`AbsoluteLayout.LayoutBounds` property. Bounds are set as a comma-separated list of values, `X`, `Y`, `Width`, and `Height`, in that order. Flags are also specified in the declaration of views in the layout using the `AbsoluteLayout.LayoutFlags` property. Note that flags can be combined in XAML using a comma-separated list. Consider the following example:

```
<?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="LayoutSamples.AbsoluteLayoutExploration"
    Title="Absolute Layout Exploration">
    <ContentPage.Content>
        <AbsoluteLayout>
            <Label Text="I'm centered on iPhone 4 but no other device"
                AbsoluteLayout.LayoutBounds="115,150,100,100" LineBreakMode="WordWrap" />
            <Label Text="I'm bottom center on every device."
                AbsoluteLayout.LayoutBounds=".5,1,.5,.1" AbsoluteLayout.LayoutFlags="All"
                LineBreakMode="WordWrap" />
            <BoxView Color="Olive" AbsoluteLayout.LayoutBounds="1,.5, 25, 100"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
            <BoxView Color="Red" AbsoluteLayout.LayoutBounds="0,.5,25,100"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
            <BoxView Color="Blue" AbsoluteLayout.LayoutBounds=".5,0,100,25"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>
```



Note the following:

- The label in the center is positioned using absolute size and position values. Because of that, it appears centered on iPhone 4S and lower, but not centered on larger devices.
- The text at the bottom of the layout is positioned using proportional size and position values. It will always

appear at the bottom center of the layout, but its size will grow with larger layout sizes.

- Three colored `BoxView`s are positioned at the top, left, and right edges of the screen using proportional position and absolute size.

The following achieves the same layout in C#:

```
public class AbsoluteLayoutExplorationCode : ContentPage
{
    public AbsoluteLayoutExplorationCode ()
    {
        Title = "Absolute Layout Exploration - Code";
        var layout = new AbsoluteLayout();

        var centerLabel = new Label {
            Text = "I'm centered on iPhone 4 but no other device.",
            LineBreakMode = LineBreakMode.WordWrap};

        AbsoluteLayout.SetLayoutBounds (centerLabel, new Rectangle (115, 159, 100, 100));
        // No need to set layout flags, absolute positioning is the default

        var bottomLabel = new Label { Text = "I'm bottom center on every device.", LineBreakMode =
LineBreakMode.WordWrap };
        AbsoluteLayout.SetLayoutBounds (bottomLabel, new Rectangle (.5, 1, .5, .1));
        AbsoluteLayout.SetLayoutFlags (bottomLabel, AbsoluteLayoutFlags.All);

        var rightBox = new BoxView{ Color = Color.Olive };
        AbsoluteLayout.SetLayoutBounds (rightBox, new Rectangle (1, .5, 25, 100));
        AbsoluteLayout.SetLayoutFlags (rightBox, AbsoluteLayoutFlags.PositionProportional);

        var leftBox = new BoxView{ Color = Color.Red };
        AbsoluteLayout.SetLayoutBounds (leftBox, new Rectangle (0, .5, 25, 100));
        AbsoluteLayout.SetLayoutFlags (leftBox, AbsoluteLayoutFlags.PositionProportional);

        var topBox = new BoxView{ Color = Color.Blue };
        AbsoluteLayout.SetLayoutBounds (topBox, new Rectangle (.5, 0, 100, 25));
        AbsoluteLayout.SetLayoutFlags (topBox, AbsoluteLayoutFlags.PositionProportional);

        layout.Children.Add (bottomLabel);
        layout.Children.Add (centerLabel);
        layout.Children.Add (rightBox);
        layout.Children.Add (leftBox);
        layout.Children.Add (topBox);

        Content = layout;
    }
}
```

## Proportional Values

Proportional values define a relationship between a layout and a view. This relationship defines a child view's position or scale value as a proportion of the corresponding value of the parent layout. These values are expressed as `double`s with values between 0 and 1.

Proportional values are used to position and size views within the layout. So, when a view's width is set as a proportion, the resultant width value is the proportion multiplied by the `AbsoluteLayout`'s width. For example, with an `AbsoluteLayout` of width `500` and a view set to have a proportional width of `.5`, the rendered width of the view will be `250` (`500 x .5`).

To use proportional values, set `LayoutBounds` using `(x, y)` proportions and proportional sizes, then set `LayoutFlags` to `All`.

In XAML:

```
<Label Text="I'm bottom center on every device."  
    AbsoluteLayout.LayoutBounds=".5,.1,.5,.1" AbsoluteLayout.LayoutFlags="All" />
```

In C#:

```
var label = new Label {Text = "I'm bottom center on every device."};  
AbsoluteLayout.SetLayoutBounds(label, new Rectangle(.5,.1,.5,.1));  
AbsoluteLayout.SetLayoutFlags(label, AbsoluteLayoutFlags.All);
```

## Absolute Values

Absolute values explicitly define where views should be positioned within the layout. Unlike proportional values, absolute values are capable of positioning and sizing a view that does not fit within the bounds of the layout.

Using absolute values for positioning can be dangerous when the size of the layout is not known. When using absolute positions, an element in the center of the screen at one size could be offset at any other size. It is important to test your app across the various screen sizes of your supported devices.

To use absolute layout values, set `LayoutBounds` using (x, y) coordinates and explicit sizes, then set `LayoutFlags` to `None`.

In XAML:

```
<Label Text="I'm centered on iPhone 4 but no other device."  
    AbsoluteLayout.LayoutBounds="115,150,100,100" />
```

In C#:

```
var label = new Label {Text = "I'm centered on iPhone 4 but no other device."};  
AbsoluteLayout.SetLayoutBounds(label, new Rectangle(115,150,100,100));
```

## Exploring a Complex Layout

Each of the layouts have strengths and weaknesses for creating particular layouts. Throughout this series of layout articles, a sample app has been created with the same page layout implemented using three different layouts.

Consider the following XAML:

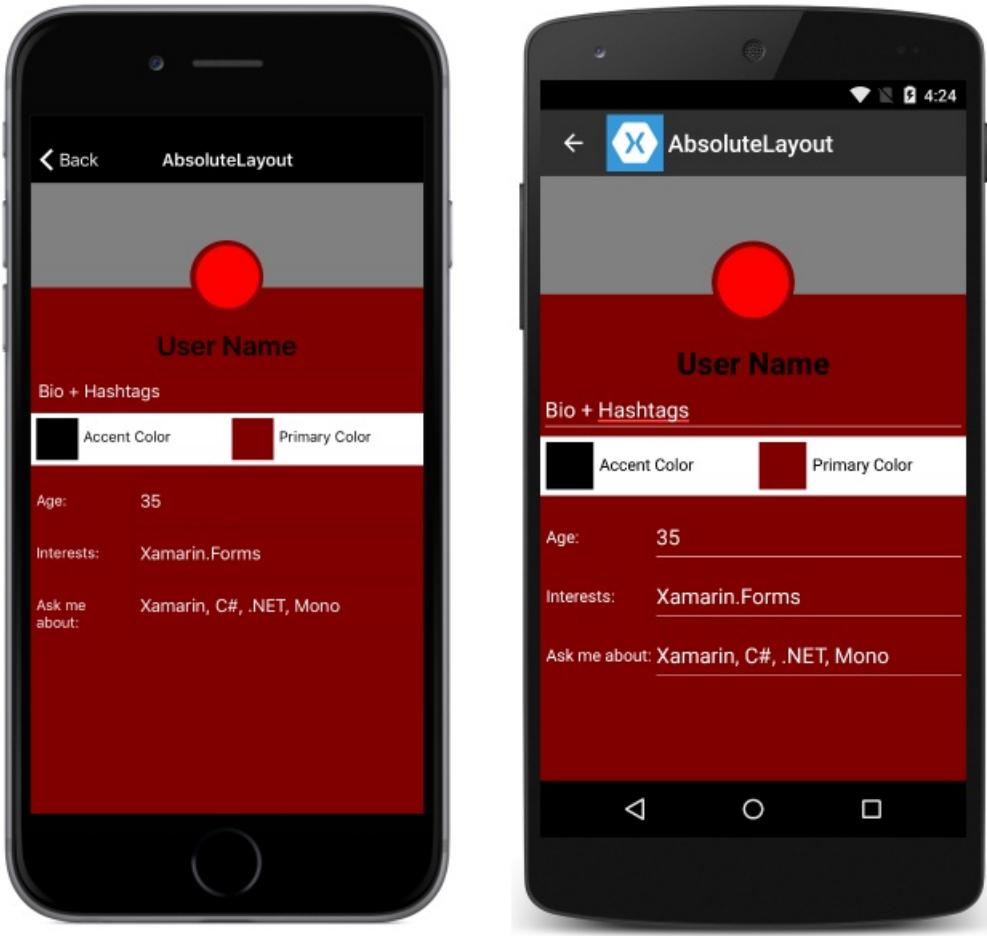
```
<?xml version="1.0" encoding="UTF-8"?>  
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"  
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"  
    x:Class="TheBusinessTumble.AbsoluteLayoutPage"  
    Title="AbsoluteLayout">  
    <ContentPage.ToolbarItems>  
        <ToolbarItem Text="Save" />  
    </ContentPage.ToolbarItems>  
    <ContentPage.Content>  
        <ScrollView>  
            <AbsoluteLayout BackgroundColor="Maroon">  
                <BoxView BackgroundColor="Gray" AbsoluteLayout.LayoutBounds="0  
                    ,1,100" AbsoluteLayout.LayoutFlags="XProportional,YProportional,WidthProportional" />  
                <Button BackgroundColor="Maroon"  
                    AbsoluteLayout.LayoutBounds=".5,55,70,70" AbsoluteLayout.LayoutFlags="XProportional"  
                    BorderRadius="35" />  
                <Button BackgroundColor="Red" AbsoluteLayout.LayoutBounds=".5  
                    ,60,60" AbsoluteLayout.LayoutFlags="XProportional" BorderRadius="30" />  
                <Label Text="User Name" FontAttributes="Bold" FontSize="26"  
                    TextColor="Black" HorizontalTextAlignment="Center" />
```

```

        AbsoluteLayout.LayoutBounds=".5,140,1,40" AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
            <Entry Text="Bio + Hashtags" TextColor="White"
                BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds=".5,180,1,40"
        AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
            <AbsoluteLayout BackgroundColor="White"
                AbsoluteLayout.LayoutBounds="0, 220, 1, 50"
        AbsoluteLayout.LayoutFlags="XProportional,WidthProportional">
            <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,0,.5,1"
        AbsoluteLayout.LayoutFlags="WidthProportional,HeightProportional">
            <Button BackgroundColor="Black" BorderRadius="20"
                AbsoluteLayout.LayoutBounds="5,.5,40,40"
        AbsoluteLayout.LayoutFlags="YProportional" />
            <Label Text="Accent Color" TextColor="Black"
                AbsoluteLayout.LayoutBounds="50,.55,1,25"
        AbsoluteLayout.LayoutFlags="YProportional,WidthProportional" />
            </AbsoluteLayout>
            <AbsoluteLayout AbsoluteLayout.LayoutBounds="1,0,.5,1"
        AbsoluteLayout.LayoutFlags="WidthProportional,HeightProportional,XProportional">
            <Button BackgroundColor="Maroon" BorderRadius="20"
                AbsoluteLayout.LayoutBounds="5,.5,40,40"
        AbsoluteLayout.LayoutFlags="YProportional" />
            <Label Text="Primary Color" TextColor="Black"
                AbsoluteLayout.LayoutBounds="50,.55,1,25"
        AbsoluteLayout.LayoutFlags="YProportional,WidthProportional" />
            </AbsoluteLayout>
            </AbsoluteLayout>
            <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,270,1,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
            <Label Text="Age:" TextColor="White"
                AbsoluteLayout.LayoutBounds="0,25,.25,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" />
            <Entry Text="35" TextColor="White" BackgroundColor="Maroon"
                AbsoluteLayout.LayoutBounds="1,10,.75,50"
        AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
            </AbsoluteLayout>
            <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,320,1,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
            <Label Text="Interests:" TextColor="White"
                AbsoluteLayout.LayoutBounds="0,25,.25,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" />
            <Entry Text="Xamarin.Forms" TextColor="White"
                BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds="1,10,.75,50"
        AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
            </AbsoluteLayout>
            <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,370,1,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
            <Label Text="Ask me about:" TextColor="White"
                AbsoluteLayout.LayoutBounds="0,25,.25,50"
        AbsoluteLayout.LayoutFlags="WidthProportional" />
            <Entry Text="Xamarin, C#, .NET, Mono" TextColor="White"
                BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds="1,10,.75,50"
        AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
            </AbsoluteLayout>
            </AbsoluteLayout>
        </ScrollView>
    </ContentPage.Content>
</ContentPage>

```

The above code results in the following layout:



Notice that `AbsoluteLayout`s are nested, because in some cases nesting layouts can be easier than presenting all elements within the same layout.

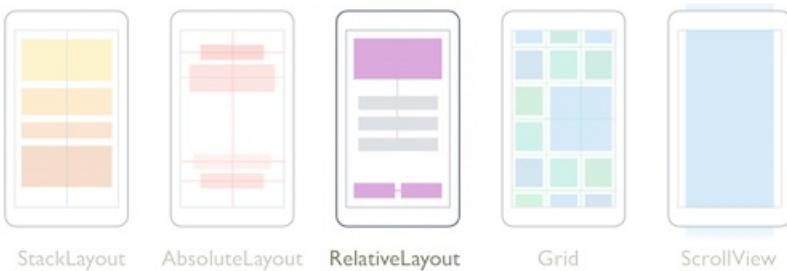
## Related Links

- [Creating Mobile Apps with Xamarin.Forms, Chapter 14](#)
- [AbsoluteLayout](#)
- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# Xamarin.Forms RelativeLayout

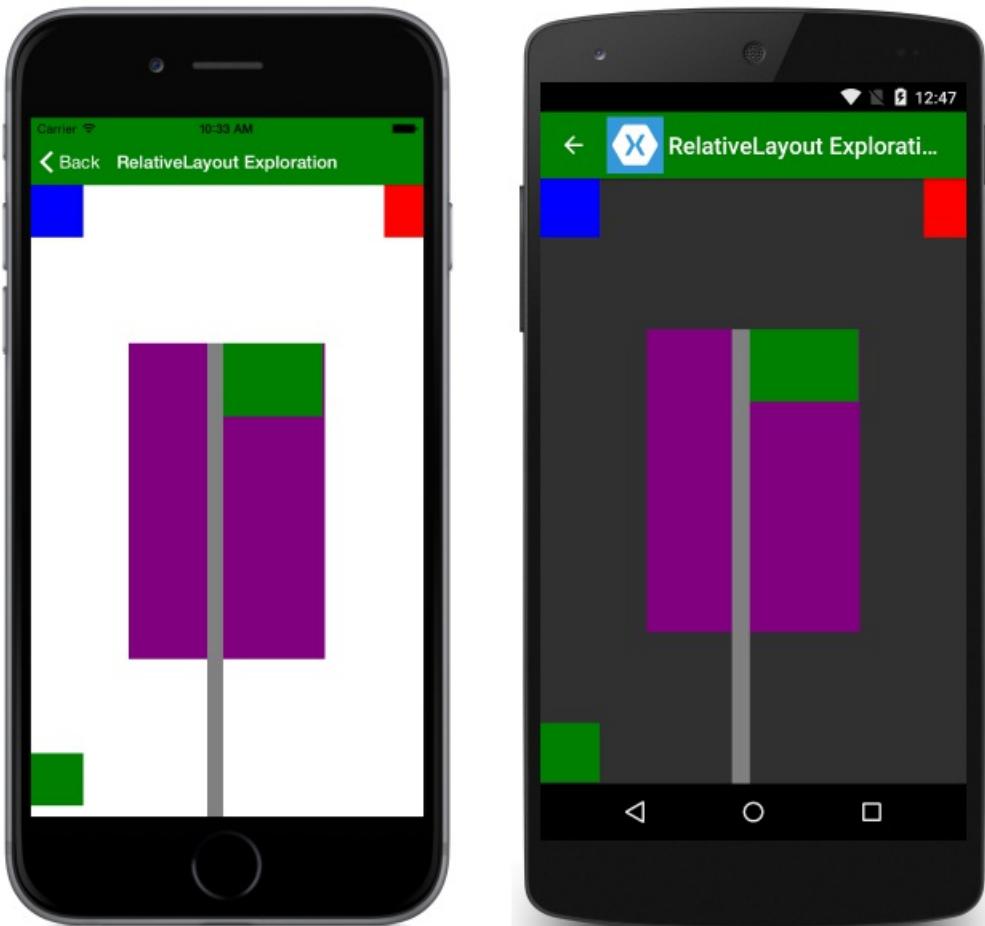
6/8/2018 • 5 minutes to read • [Edit Online](#)

`RelativeLayout` is used to position and size views relative to properties of the layout or sibling views. Unlike `AbsoluteLayout`, `RelativeLayout` does not have the concept of the moving anchor and does not have facilities for positioning elements relative to the bottom or right edges of the layout. `RelativeLayout` does support positioning elements outside of its own bounds.



## Purpose

`RelativeLayout` can be used to position views on screen relative to the overall layout or to other views.



## Usage

### Understanding Constraints

Positioning and sizing a view within a `RelativeLayout` is done with constraints. A constraint expression can

include the following information:

- **Type** – whether the constraint is relative to the parent or to another view.
- **Property** – which property to use as the basis for the constraint.
- **Factor** – the factor to apply to the property value.
- **Constant** – the value to use as an offset of the value.
- **ElementName** – the name of the view that the constraint is relative to.

In XAML, constraints are expressed as `ConstraintExpression`s. Consider the following example:

```
<BoxView Color="Green" WidthRequest="50" HeightRequest="50"
    RelativeLayout.XConstraint =
        "{ConstraintExpression Type=RelativeToParent,
            Property=Width,
            Factor=0.5,
            Constant=-100}"
    RelativeLayout.YConstraint =
        "{ConstraintExpression Type=RelativeToParent,
            Property=Height,
            Factor=0.5,
            Constant=-100}" />
```

In C#, constraints are expressed a little differently, using functions rather than expressions on the view. Constraints are specified as arguments to the layout's `Add` method:

```
layout.Children.Add(box, Constraint.RelativeToParent((parent) =>
{
    return (.5 * parent.Width) - 100;
}),
Constraint.RelativeToParent((parent) =>
{
    return (.5 * parent.Height) - 100;
}),
Constraint.Constant(50), Constraint.Constant(50));
```

Note the following aspects of the above layout:

- The `x` and `y` constraints are specified with their own constraints.
- In C#, relative constraints are defined as functions. Concepts like `Factor` aren't there, but can be implemented manually.
- The box's `x` coordinate is defined as half the width of the parent, -100.
- The box's `y` coordinate is defined as half the height of the parent, -100.

#### NOTE

Because of the way constraints are defined, it is possible to make more complex layouts in C# than can be specified with XAML.

Both of the examples above define constraints as `RelativeToParent` – that is, their values are relative to the parent element. It is also possible to define constraints as relative to another view. This allows for more intuitive (to the developer) layouts and can make the intent of your layout code more readily apparent.

Consider a layout where one element needs to be 20 pixels lower than another. If both elements are defined with constant values, the lower could have its `y` constraint defined as a constant that is 20 pixels greater than the `y` constraint of the higher element. That approach falls short if the higher element is positioned using a proportion, so that the pixel size isn't known. In that case, constraining the element based on another element's position is

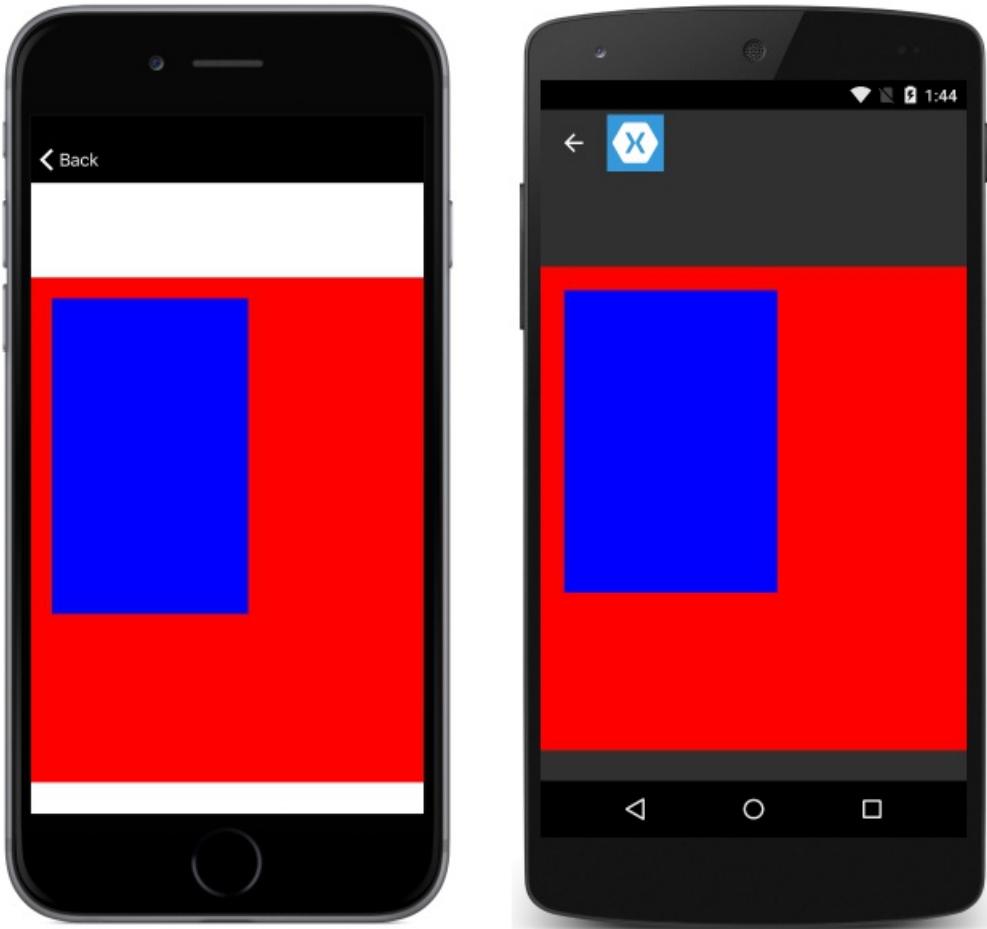
more robust:

```
<RelativeLayout>
    <BoxView Color="Red" x:Name="redBox"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
            Property=Height,Factor=.15,Constant=0}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=1,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.8,Constant=0}" />
    <BoxView Color="Blue"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToView,
            ElementName=redBox,Property=Y,Factor=1,Constant=20}"
        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView,
            ElementName=redBox,Property=X,Factor=1,Constant=20}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.5,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.5,Constant=0}" />
    </RelativeLayout>
```

To accomplish the same layout in C#:

```
layout.Children.Add (redBox, Constraint.RelativeToParent ((parent) => {
    return parent.X;
}), Constraint.RelativeToParent ((parent) => {
    return parent.Y * .15;
}), Constraint.RelativeToParent((parent) => {
    return parent.Width;
}), Constraint.RelativeToParent((parent) => {
    return parent.Height * .8;
}));  
layout.Children.Add (blueBox, Constraint.RelativeToView (redBox, (Parent, sibling) => {
    return sibling.X + 20;
}), Constraint.RelativeToView (blueBox, (parent, sibling) => {
    return sibling.Y + 20;
}), Constraint.RelativeToParent((parent) => {
    return parent.Width * .5;
}), Constraint.RelativeToParent((parent) => {
    return parent.Height * .5;
}));
```

This produces the following output, with the blue box's position determined *relative* to the position of the red box:



## Sizing

Views laid out by `RelativeLayout` have two options for specifying their size:

- `HeightRequest` & `WidthRequest`
- `RelativeLayout.WidthConstraint` & `RelativeLayout.HeightConstraint`

`HeightRequest` and `WidthRequest` specify the intended height and width of the view, but may be overridden by layouts as needed. `WidthConstraint` and `HeightConstraint` support setting the height and width as a value relative to the layout's or another view's properties, or as a constant value.

## Exploring a Complex Layout

Each of the layouts have strengths and weaknesses for creating particular layouts. Throughout this series of layout articles, a sample app has been created with the same page layout implemented using three different layouts.

Consider the following XAML:

```
<?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="TheBusinessTumble.RelativeLayoutPage"
  BackgroundColor="Maroon"
  Title="RelativeLayout">
  <ContentPage.Content>
    <ScrollView>
      <RelativeLayout>
        <BoxView Color="Gray" HeightRequest="100"
          RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
          Factor=1}" />
        <Button BorderRadius="35" x:Name="imageCircleBack"
          BackgroundColor="Maroon" HeightRequest="70" WidthRequest="70" RelativeLayout.XConstraint="
          {ConstraintExpression Type=RelativeToParent, Property=Width, Factor= 5, Constant = -351}"
```

```

<ConstraintExpression Type=RelativeToParent, Property=Width, Factor=.5, Constant=-50>
RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Factor=0, Property=Y, Constant=70}"
/>

    <Button BorderRadius="30" BackgroundColor="Red" HeightRequest="60"
        WidthRequest="60" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView,
ElementName=imageCircleBack, Property=X, Factor=1,Constant=5}" RelativeLayout.YConstraint=
{ConstraintExpression Type=RelativeToParent, Factor=0, Property=Y, Constant=75}" />
    <Label Text="User Name" FontAttributes="Bold" FontSize="26"
        HorizontalTextAlignment="Center" RelativeLayout.YConstraint="{ConstraintExpression
Type=RelativeToParent, Property=Y, Factor=0, Constant=140}" RelativeLayout.WidthConstraint=
{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=1}" />
    <Entry Text="Bio + Hashtags" TextColor="White" BackgroundColor="Maroon"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0,
Constant=180}" RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=1}" />
    <RelativeLayout BackgroundColor="White" RelativeLayout.YConstraint=
{ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0, Constant=220}"
HeightRequest="60" RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=1}" >
        <BoxView BackgroundColor="Black" WidthRequest="50"
            HeightRequest="50" RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Y, Factor=0, Constant=5}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
Property=X, Factor=0, Constant=5}" />
        <BoxView BackgroundColor="Maroon" WidthRequest="50"
            HeightRequest="50" RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Y, Factor=0, Constant=5}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=0.5, Constant=}" />
        <Label FontSize="14" TextColor="Black" Text="Accent Color"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0,
Constant=20}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=X, Factor=0,
Constant=60}" />
        <Label FontSize="14" TextColor="Black" Text="Primary Color"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0,
Constant=20}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.5, Constant=55}" />
    </RelativeLayout>
    <RelativeLayout Padding="5,0,0,0">
        <Label FontSize="14" Text="Age:" TextColor="White"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=305}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=0,
Constant=10}">
            <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=.25, Constant=0}">
                <Label FontSize="14" Text="Interests:" TextColor="White"
                    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=280}"
                    RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.3, Constant=0}">
                    <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=0.75, Constant=0}">
                        <Label FontSize="14" Text="Xamarin.Forms" TextColor="White" BackgroundColor="Maroon"
                            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=320}">
                            <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=.25, Constant=0}">
                                <Label FontSize="14" Text="Interests:" TextColor="White"
                                    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=345}">
                                    <RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}">
                                        <Label FontSize="14" Text="Xamarin.Forms" TextColor="White" BackgroundColor="Maroon"
                                            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=320}">
                                            <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=.25, Constant=0}">
                                                <Label FontSize="14" Text="Interests:" TextColor="White"
                                                    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=345}">
                                                    <RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}">
                                                        <Label FontSize="14" Text="Xamarin.Forms" TextColor="White" BackgroundColor="Maroon"
                                                            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=320}">
                                                            <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=.25, Constant=0}">
                                                                <Label FontSize="14" Text="Interests:" TextColor="White"
                                                                    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=345}">
                                                                    <RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}">
                                                                        <Label FontSize="14" Text="Xamarin.Forms" TextColor="White" BackgroundColor="Maroon"
                                                                            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=320}">
                                                                            <RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=.25, Constant=0}">

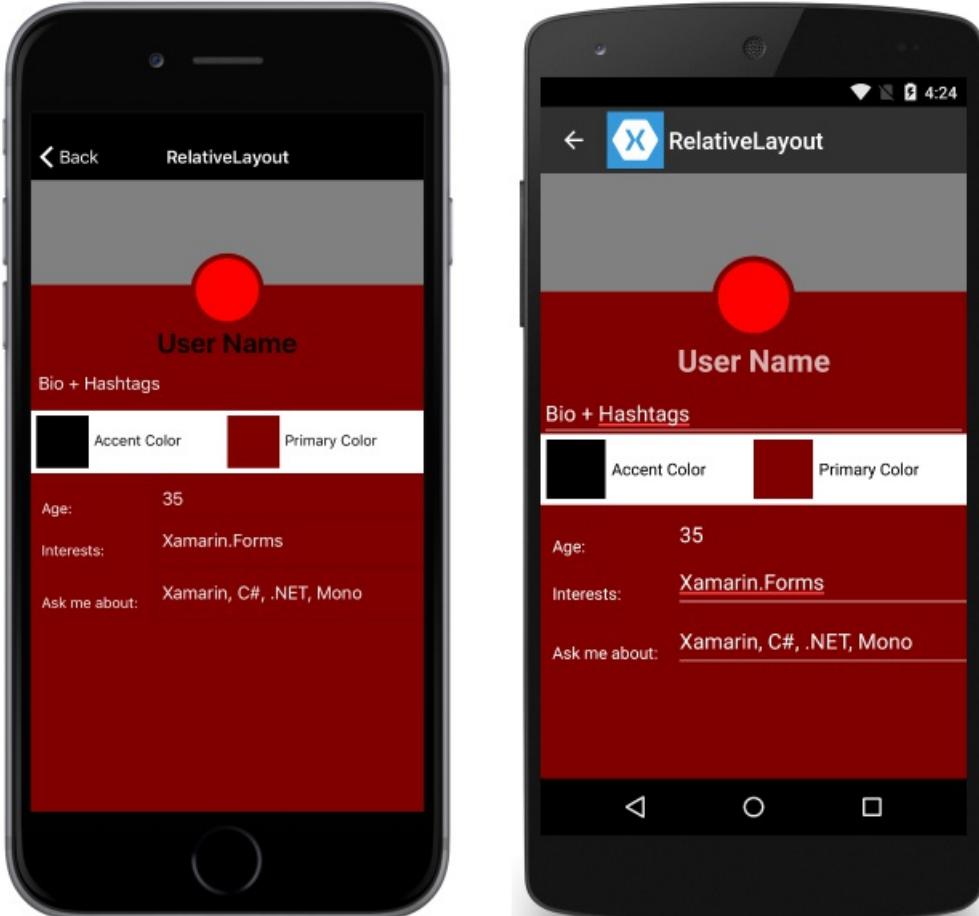
```

```

        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.3, Constant=0}"
        RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=0.75, Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=50}"/>
    </RelativeLayout>
    <RelativeLayout Padding="5,0,0,0">
        <Label FontSize="14" Text="Ask me about:" TextColor="White"
LineBreakMode="WordWrap"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=395}"
        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=0,
Constant=10}"
        RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=.25, Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=50}"/>
            <Entry Text="Xamarin, C#, .NET, Mono" TextColor="White"
BackgroundColor="Maroon"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=370}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=0.3, Constant=0}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=0.75, Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0, Constant=50}"/>
                </Entry>
            </RelativeLayout>
        </RelativeLayout>
    </ScrollView>
</ContentPage.Content>
</ContentPage>

```

The above code results in the following layout:



Notice that `RelativeLayout`s are nested, because in some cases nesting layouts can be easier than presenting all elements within the same layout. Also notice that some elements are `RelativeLayout`, because that allows for easier and more intuitive layout when the relationships between views guide positioning.

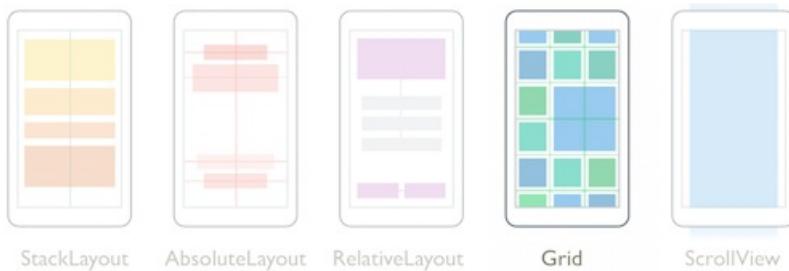
## Related Links

- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# Xamarin.Forms Grid

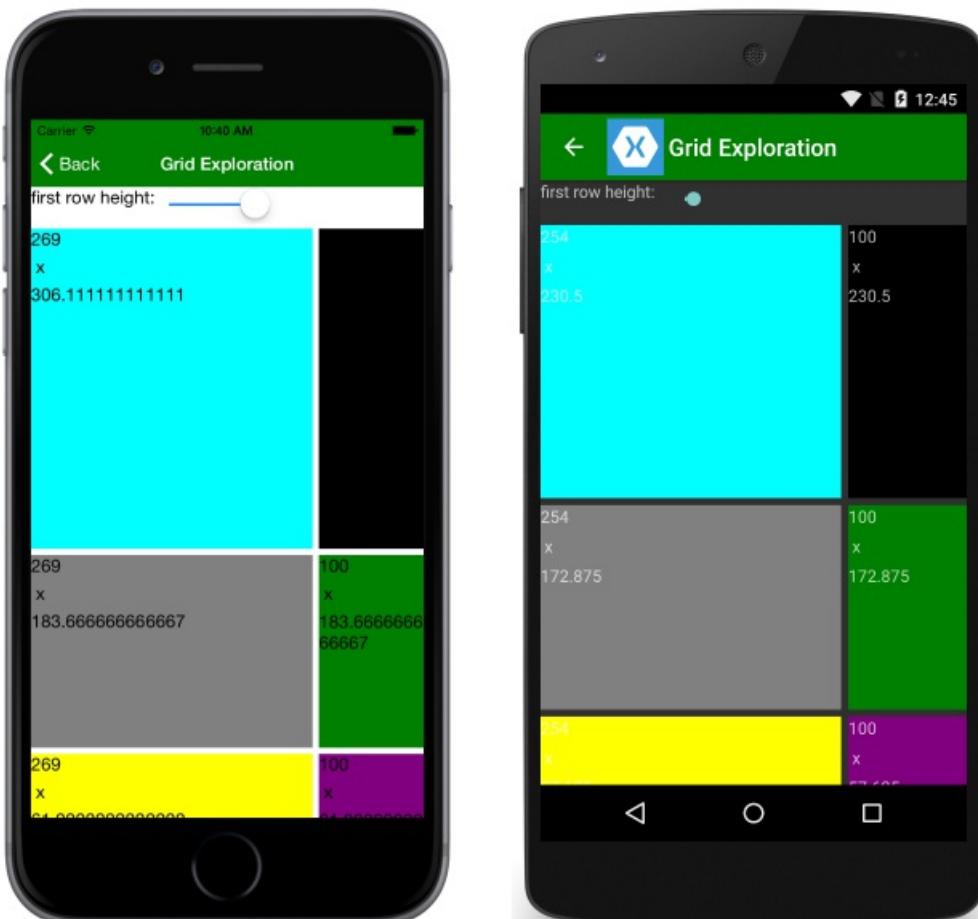
11/21/2018 • 10 minutes to read • [Edit Online](#)

`Grid` supports arranging views into rows and columns. Rows and columns can be set to have proportional sizes or absolute sizes. The `Grid` layout should not be confused with traditional tables and is not intended to present tabular data. `Grid` does not have the concept of row, column or cell formatting. Unlike HTML tables, `Grid` is purely intended for laying out content.



This article will cover:

- **Purpose** – common uses for `Grid`.
- **Usage** – how to use `Grid` to achieve your desired design.
  - **Rows and Columns** – specify rows and columns for the `Grid`.
  - **Placing Views** – add views to the grid at specific rows and columns.
  - **Spacing** – configure the spaces between rows and columns.
  - **Spans** – configure elements to span across multiple rows or columns.



# Purpose

`Grid` can be used to arrange views into a grid. This is useful in a number of cases:

- Arranging buttons in a calculator app
- Arranging buttons/choices in a grid, like the iOS or Android home screens
- Arranging views so that they are of equal size in one dimension ( like in some toolbars)

# Usage

Unlike traditional tables, `Grid` does not infer the number and sizes of rows and columns from the content. Instead, `Grid` has `RowDefinitions` and `ColumnDefinitions` collections. These hold definitions of how many rows and columns will be laid out. Views are added to `Grid` with specified row and column indices, which identify which row and column a view should be placed in.

## Rows and Columns

Row and column information is stored in `Grid`'s `RowDefinitions` & `ColumnDefinitions` properties, which are each collections of `RowDefinition` and `ColumnDefinition` objects, respectively. `RowDefinition` has a single property, `Height`, and `ColumnDefinition` has a single property, `Width`. The options for height and width are as follows:

- **Auto** – automatically sizes to fit content in the row or column. Specified as `GridUnitType.Auto` in C# or as `Auto` in XAML.
- **Proportional(\*)** – sizes columns and rows as a proportion of the remaining space. Specified as a value and `GridUnitType.Star` in C# and as `#*` in XAML, with `#` being your desired value. Specifying one row/column with `*` will cause it to fill the available space.
- **Absolute** – sizes columns and rows with specific, fixed height and width values. Specified as a value and `GridUnitType.Absolute` in C# and as `#` in XAML, with `#` being your desired value.

### NOTE

The width values for columns are set as `\*` by default in Xamarin.Forms, which ensures that the column will fill the available space.

Consider an app that needs three rows and two columns. The bottom row needs to be exactly 200px tall and the top row needs to be twice as tall as the middle row. The left column needs to be wide enough to fit the content and the right column needs to fill the remaining space.

In XAML:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="2*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="200" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
</Grid>
```

In C#:

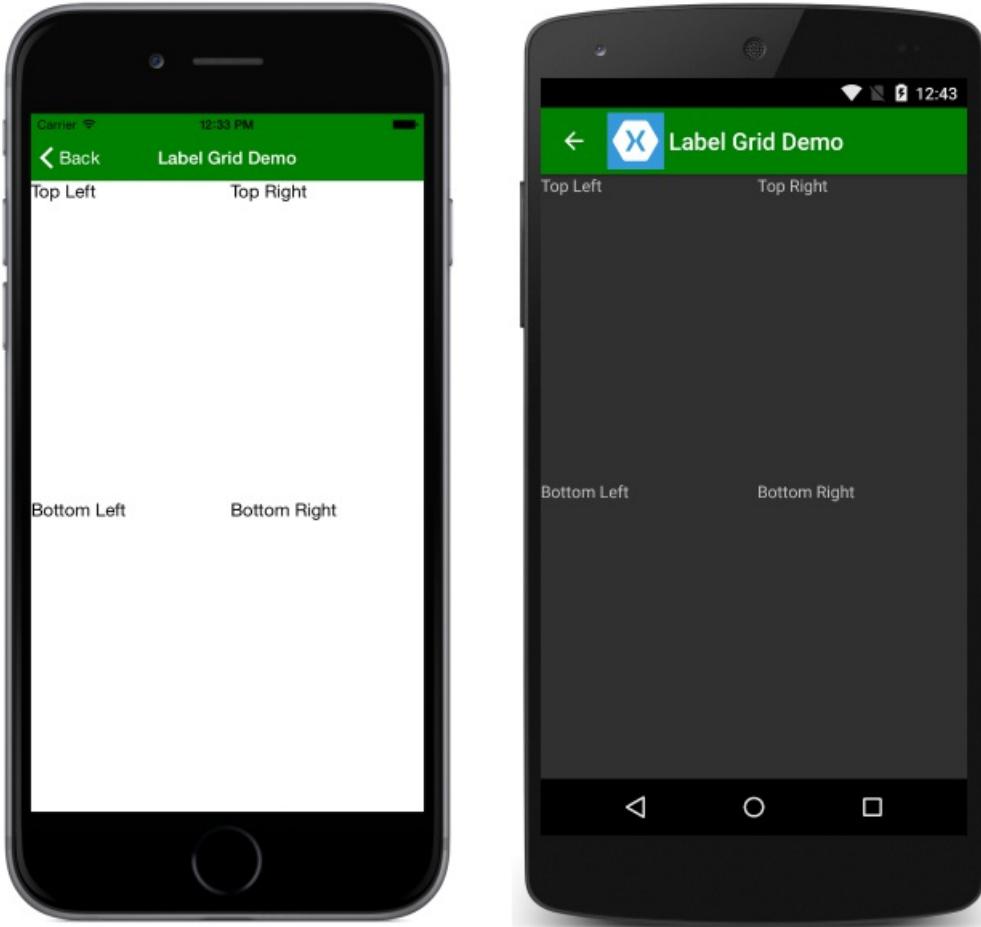
```
var grid = new Grid();
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength(2, GridUnitType.Star) });
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength(200)} );
grid.ColumnDefinitions.Add (new ColumnDefinition{ Width = new GridLength (200) });
```

## Placing Views in a Grid

To place views in a `Grid` you'll need to add them as children to the grid, then specify which row and column they belong in.

In XAML, use `Grid.Row` and `Grid.Column` on each individual view to specify placement. Note that `Grid.Row` and `Grid.Column` specify location based on the zero-based lists of rows and columns. This means that in a 4x4 grid, the top left cell is (0,0) and the bottom right cell is (3,3).

The `Grid` shown below contains four cells:



In XAML:

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Text="Top Left" Grid.Row="0" Grid.Column="0" />
  <Label Text="Top Right" Grid.Row="0" Grid.Column="1" />
  <Label Text="Bottom Left" Grid.Row="1" Grid.Column="0" />
  <Label Text="Bottom Right" Grid.Row="1" Grid.Column="1" />
</Grid>

```

In C#:

```

var grid = new Grid();

grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(1, GridUnitType.Star)});
grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(1, GridUnitType.Star)});
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(1, GridUnitType.Star)});
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(1, GridUnitType.Star)});

var topLeft = new Label { Text = "Top Left" };
var topRight = new Label { Text = "Top Right" };
var bottomLeft = new Label { Text = "Bottom Left" };
var bottomRight = new Label { Text = "Bottom Right" };

grid.Children.Add(topLeft, 0, 0);
grid.Children.Add(topRight, 1, 0);
grid.Children.Add(bottomLeft, 0, 1);
grid.Children.Add(bottomRight, 1, 1);

```

The above code creates grid with four labels, two columns, and two rows. Note that each label will have the same size and that the rows will expand to use all available space.

In the example above, views are added to the `Grid.Children` collection using the `Add` overload that specifies left and top arguments. When using the `Add` overload that specifies left, right, top, and bottom arguments, while the left and top arguments will always refer to cells within the `Grid`, the right and bottom arguments may appear to refer to cells that are outside the `Grid`. This is because the right argument must always be greater than the left argument, and the bottom argument must always be greater than the top argument. The following example shows equivalent code using both `Add` overloads:

```

// left, top
grid.Children.Add(topLeft, 0, 0);
grid.Children.Add(topRight, 1, 0);
grid.Children.Add(bottomLeft, 0, 1);
grid.Children.Add(bottomRight, 1, 1);

// left, right, top, bottom
grid.Children.Add(topLeft, 0, 1, 0, 1);
grid.Children.Add(topRight, 1, 2, 0, 1);
grid.Children.Add(bottomLeft, 0, 1, 1, 2);
grid.Children.Add(bottomRight, 1, 2, 1, 2);

```

## Spacing

`Grid` has properties to control spacing between rows and columns. The following properties are available for customizing the `Grid`:

- **ColumnSpacing** – the amount of space between columns. The default value of this property is 6.
- **RowSpacing** – the amount of space between rows. The default value of this property is 6.

The following XAML specifies a `Grid` with two columns, one row, and 5 px of spacing between columns:

```
<Grid ColumnSpacing="5">
  <Grid.ColumnDefinitions>
    <ColumnDefinitions Width="*" />
    <ColumnDefinitions Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

In C#:

```
var grid = new Grid { ColumnSpacing = 5 };
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star)} );
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star)} );
```

## Spans

Often when working with a grid, there is an element that should occupy more than one row or column. Consider a simple calculator application:



Notice that the 0 button spans two columns, just like on the built-in calculators for each platform. This is accomplished using the `ColumnSpan` property, which specifies how many columns an element should occupy. The XAML for that button:

```
<Button Text = "0" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2" />
```

And in C#:

```
Button zeroButton = new Button { Text = "0" };
controlGrid.Children.Add (zeroButton, 0, 4);
Grid.SetColumnSpan (zeroButton, 2);
```

Note that in code, static methods of the `Grid` class are used to perform positioning changes including changes to `ColumnSpan` and `RowSpan`. Also note that, unlike other properties that can be set at any time, properties set using the static methods must already be in the grid before they are changed.

The complete XAML for the above calculator app is as follows:

```
<?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="LayoutSamples.CalculatorGridXAML"
  Title = "Calculator - XAML"
  BackgroundColor="#404040">
  <ContentPage.Resources>
    <ResourceDictionary>
      <Style x:Key="plainButton" TargetType="Button">
        <Setter Property="BackgroundColor" Value="#eee"/>
        <Setter Property="TextColor" Value="Black" />
        <Setter Property="BorderRadius" Value="0"/>
        <Setter Property="FontSize" Value="40" />
      </Style>
      <Style x:Key="darkerButton" TargetType="Button">
        <Setter Property="BackgroundColor" Value="#ddd"/>
        <Setter Property="TextColor" Value="Black" />
        <Setter Property="BorderRadius" Value="0"/>
        <Setter Property="FontSize" Value="40" />
      </Style>
      <Style x:Key="orangeButton" TargetType="Button">
        <Setter Property="BackgroundColor" Value="#E8AD00"/>
        <Setter Property="TextColor" Value="White" />
        <Setter Property="BorderRadius" Value="0"/>
        <Setter Property="FontSize" Value="40" />
      </Style>
    </ResourceDictionary>
  </ContentPage.Resources>
  <ContentPage.Content>
    <Grid x:Name="controlGrid" RowSpacing="1" ColumnSpacing="1">
      <Grid.RowDefinitions>
        <RowDefinition Height="150" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Text="0" Grid.Row="0" HorizontalTextAlignment="End" VerticalTextAlignment="End"
        TextColor="White"
        FontSize="60" Grid.ColumnSpan="4" />
        <Button Text = "C" Grid.Row="1" Grid.Column="0"
          Style="{StaticResource darkerButton}" />
        <Button Text = "+/-" Grid.Row="1" Grid.Column="1"
          Style="{StaticResource darkerButton}" />
        <Button Text = "%" Grid.Row="1" Grid.Column="2"
          Style="{StaticResource darkerButton}" />
```

```

        <Button Text = "div" Grid.Row="1" Grid.Column="3"
Style="{StaticResource orangeButton}" />
        <Button Text = "7" Grid.Row="2" Grid.Column="0"
Style="{StaticResource plainButton}" />
        <Button Text = "8" Grid.Row="2" Grid.Column="1"
Style="{StaticResource plainButton}" />
        <Button Text = "9" Grid.Row="2" Grid.Column="2"
Style="{StaticResource plainButton}" />
        <Button Text = "X" Grid.Row="2" Grid.Column="3"
Style="{StaticResource orangeButton}" />
        <Button Text = "4" Grid.Row="3" Grid.Column="0"
Style="{StaticResource plainButton}" />
        <Button Text = "5" Grid.Row="3" Grid.Column="1"
Style="{StaticResource plainButton}" />
        <Button Text = "6" Grid.Row="3" Grid.Column="2"
Style="{StaticResource plainButton}" />
        <Button Text = "-" Grid.Row="3" Grid.Column="3"
Style="{StaticResource orangeButton}" />
        <Button Text = "1" Grid.Row="4" Grid.Column="0"
Style="{StaticResource plainButton}" />
        <Button Text = "2" Grid.Row="4" Grid.Column="1"
Style="{StaticResource plainButton}" />
        <Button Text = "3" Grid.Row="4" Grid.Column="2"
Style="{StaticResource plainButton}" />
        <Button Text = "+" Grid.Row="4" Grid.Column="3"
Style="{StaticResource orangeButton}" />
        <Button Text = "0" Grid.ColumnSpan="2"
Grid.Row="5" Grid.Column="0" Style="{StaticResource plainButton}" />
        <Button Text = "." Grid.Row="5" Grid.Column="2"
Style="{StaticResource plainButton}" />
        <Button Text = "=" Grid.Row="5" Grid.Column="3"
Style="{StaticResource orangeButton}" />
    </Grid>
</ContentPage.Content>
</ContentPage>

```

Notice that both the label at the top of the grid and the zero button are occupying more than one column. Although a similar layout could be achieved using nested grids, the `ColumnSpan` & `RowSpan` approach is simpler.

The C# implementation:

```

public CalculatorGridCode ()
{
    Title = "Calculator - C#";
    BackgroundColor = Color.FromHex ("#404040");

    var plainButton = new Style (typeof(Button)) {
        Setters = {
            new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#eee") },
            new Setter { Property = Button.TextColorProperty, Value = Color.Black },
            new Setter { Property = Button.BorderRadiusProperty, Value = 0 },
            new Setter { Property = Button.FontSizeProperty, Value = 40 }
        }
    };
    var darkerButton = new Style (typeof(Button)) {
        Setters = {
            new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#ddd") },
            new Setter { Property = Button.TextColorProperty, Value = Color.Black },
            new Setter { Property = Button.BorderRadiusProperty, Value = 0 },
            new Setter { Property = Button.FontSizeProperty, Value = 40 }
        }
    };
    var orangeButton = new Style (typeof(Button)) {
        Setters = {
            new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#E8AD00") },
            new Setter { Property = Button.TextColorProperty, Value = Color.White },
            new Setter { Property = Button.BorderRadiusProperty, Value = 0 }
        }
    };
}

```

```

        new Setter { Property = Button.FontSizeProperty, Value = 40 }
    }

};

var controlGrid = new Grid { RowSpacing = 1, ColumnSpacing = 1 };
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (150) });
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });

controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });

var label = new Label {
    Text = "0",
    HorizontalTextAlignment = TextAlignment.End,
    VerticalTextAlignment = TextAlignment.End,
    TextColor = Color.White,
    FontSize = 60
};
controlGrid.Children.Add (label, 0, 0);

Grid.SetColumnSpan (label, 4);

controlGrid.Children.Add (new Button { Text = "C", Style = darkerButton }, 0, 1);
controlGrid.Children.Add (new Button { Text = "+/-", Style = darkerButton }, 1, 1);
controlGrid.Children.Add (new Button { Text = "%", Style = darkerButton }, 2, 1);
controlGrid.Children.Add (new Button { Text = "div", Style = orangeButton }, 3, 1);
controlGrid.Children.Add (new Button { Text = "7", Style = plainButton }, 0, 2);
controlGrid.Children.Add (new Button { Text = "8", Style = plainButton }, 1, 2);
controlGrid.Children.Add (new Button { Text = "9", Style = plainButton }, 2, 2);
controlGrid.Children.Add (new Button { Text = "X", Style = orangeButton }, 3, 2);
controlGrid.Children.Add (new Button { Text = "4", Style = plainButton }, 0, 3);
controlGrid.Children.Add (new Button { Text = "5", Style = plainButton }, 1, 3);
controlGrid.Children.Add (new Button { Text = "6", Style = plainButton }, 2, 3);
controlGrid.Children.Add (new Button { Text = "-", Style = orangeButton }, 3, 3);
controlGrid.Children.Add (new Button { Text = "1", Style = plainButton }, 0, 4);
controlGrid.Children.Add (new Button { Text = "2", Style = plainButton }, 1, 4);
controlGrid.Children.Add (new Button { Text = "3", Style = plainButton }, 2, 4);
controlGrid.Children.Add (new Button { Text = "+", Style = orangeButton }, 3, 4);
controlGrid.Children.Add (new Button { Text = ".", Style = plainButton }, 2, 5);
controlGrid.Children.Add (new Button { Text = "=", Style = orangeButton }, 3, 5);

var zeroButton = new Button { Text = "0", Style = plainButton };
controlGrid.Children.Add (zeroButton, 0, 5);
Grid.SetColumnSpan (zeroButton, 2);

Content = controlGrid;
}

```

## Related Links

- [Creating Mobile Apps with Xamarin.Forms, Chapter 17](#)
- [Grid](#)
- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# The Xamarin.Forms FlexLayout

11/20/2018 • 23 minutes to read • [Edit Online](#)

Use `FlexLayout` for stacking or wrapping a collection of child views.

The Xamarin.Forms `FlexLayout` is new in Xamarin.Forms version 3.0. It is based on the CSS [Flexible Box Layout Module](#), commonly known as *flex layout* or *flex-box*, so called because it includes many flexible options to arrange children within the layout.

`FlexLayout` is similar to the Xamarin.Forms `StackLayout` in that it can arrange its children horizontally and vertically in a stack. However, the `FlexLayout` is also capable of wrapping its children if there are too many to fit in a single row or column, and also has many options for orientation, alignment, and adapting to various screen sizes.

`FlexLayout` derives from `Layout<View>` and inherits a `Children` property of type `IList<View>`.

`FlexLayout` defines six public bindable properties and five attached bindable properties that affect the size, orientation, and alignment of its child elements. (If you're not familiar with attached bindable properties, see the article [Attached properties](#).) These properties are described in detail in the sections below on [The bindable properties in detail](#) and [The attached bindable properties in detail](#). However, this article begins with a section on some [Common usage scenarios](#) of `FlexLayout` that describes many of these properties more informally. Towards the end of the article, you'll see how to combine `FlexLayout` with [CSS style sheets](#).

## Common usage scenarios

The [FlexLayoutDemos](#) sample program contains several pages that demonstrate some common uses of `FlexLayout` and allows you to experiment with its properties.

### Using FlexLayout for a simple stack

The [Simple Stack](#) page shows how `FlexLayout` can substitute for a `StackLayout` but with simpler markup. Everything in this sample is defined in the XAML page. The `FlexLayout` contains four children:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:FlexLayoutDemos"
    x:Class="FlexLayoutDemos.SimpleStackPage"
    Title="Simple Stack">

    <FlexLayout Direction="Column"
        AlignItems="Center"
        JustifyContent="SpaceEvenly">

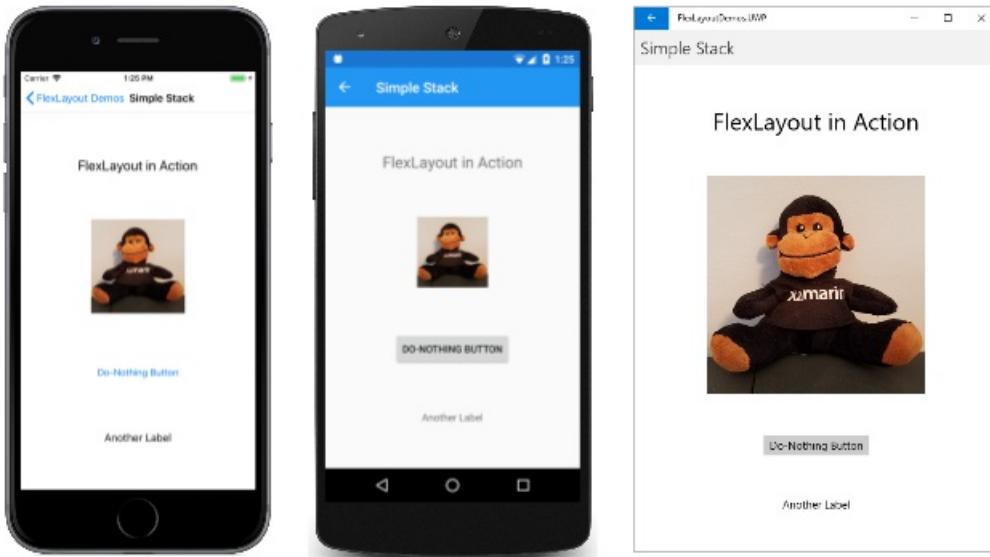
        <Label Text="FlexLayout in Action"
            FontSize="Large" />

        <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}" />

        <Button Text="Do-Nothing Button" />

        <Label Text="Another Label" />
    </FlexLayout>
</ContentPage>
```

Here's that page running on iOS, Android, and the Universal Windows Platform:



Three properties of `FlexLayout` are shown in the `SimpleStackPage.xaml` file:

- The `Direction` property is set to a value of the `FlexDirection` enumeration. The default is `Row`. Setting the property to `column` causes the children of the `FlexLayout` to be arranged in a single column of items.

When items in a `FlexLayout` are arranged in a column, the `FlexLayout` is said to have a vertical *main axis* and a horizontal *cross axis*.

- The `AlignItems` property is of type `FlexAlignItems` and specifies how items are aligned on the cross axis. The `Center` option causes each item to be horizontally centered.

If you were using a `StackLayout` rather than a `FlexLayout` for this task, you would center all the items by assigning the `HorizontalOptions` property of each item to `Center`. The `HorizontalOptions` property doesn't work for children of a `FlexLayout`, but the single `AlignItems` property accomplishes the same goal. If you need to, you can use the `AlignSelf` attached bindable property to override the `AlignItems` property for individual items:

```
<Label Text="FlexLayout in Action"
      FontSize="Large"
      FlexLayout.AlignSelf="Start" />
```

With that change, this one `Label` is positioned at the left edge of the `FlexLayout` when the reading order is left-to-right.

- The `JustifyContent` property is of type `FlexJustify`, and specifies how items are arranged on the main axis. The `SpaceEvenly` option allocates all leftover vertical space equally between all the items, and above the first item, and below the last item.

If you were using a `StackLayout`, you would need to assign the `VerticalOptions` property of each item to `CenterAndExpand` to achieve a similar effect. But the `centerAndExpand` option would allocate twice as much space between each item than before the first item and after the last item. You can mimic the `CenterAndExpand` option of `VerticalOptions` by setting the `JustifyContent` property of `FlexLayout` to `SpaceAround`.

These `FlexLayout` properties are discussed in more detail in the section [The bindable properties in detail](#) below.

### Using FlexLayout for wrapping items

The [Photo Wrapping](#) page of the [FlexLayoutDemos](#) sample demonstrates how `FlexLayout` can wrap its

children to additional rows or columns. The XAML file instantiates the `FlexLayout` and assigns two properties of it:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlexLayoutDemos.PhotoWrappingPage"
    Title="Photo Wrapping">
<Grid>
    <ScrollView>
        <FlexLayout x:Name="flexLayout"
            Wrap="Wrap"
            JustifyContent="SpaceAround" />
    </ScrollView>

    <ActivityIndicator x:Name="activityIndicator"
        IsRunning="True"
        VerticalOptions="Center" />
</Grid>
</ContentPage>
```

The `Direction` property of this `FlexLayout` is not set, so it has the default setting of `Row`, meaning that the children are arranged in rows and the main axis is horizontal.

The `Wrap` property is of an enumeration type `FlexWrap`. If there are too many items to fit on a row, then this property setting causes the items to wrap to the next row.

Notice that the `FlexLayout` is a child of a `ScrollView`. If there are too many rows to fit on the page, then the `ScrollView` has a default `Orientation` property of `Vertical` and allows vertical scrolling.

The `JustifyContent` property allocates leftover space on the main axis (the horizontal axis) so that each item is surrounded by the same amount of blank space.

The code-behind file accesses a collection of sample photos and adds them to the `Children` collection of the `FlexLayout`:

```

public partial class PhotoWrappingPage : ContentPage
{
    // Class for deserializing JSON list of sample bitmaps
    [DataContract]
    class ImageList
    {
        [DataMember(Name = "photos")]
        public List<string> Photos = null;
    }

    public PhotoWrappingPage ()
    {
        InitializeComponent ();

        LoadBitmapCollection();
    }

    async void LoadBitmapCollection()
    {
        int imageDimension = Device.RuntimePlatform == Device.iOS ||
            Device.RuntimePlatform == Device.Android ? 240 : 120;

        string urlSuffix = String.Format("?width={0}&height={0}&mode=max", imageDimension);

        using (WebClient webClient = new WebClient())
        {
            try
            {
                // Download the list of stock photos
                Uri uri = new Uri("http://docs.xamarin.com/demo/stock.json");
                byte[] data = await webClient.DownloadDataTaskAsync(uri);

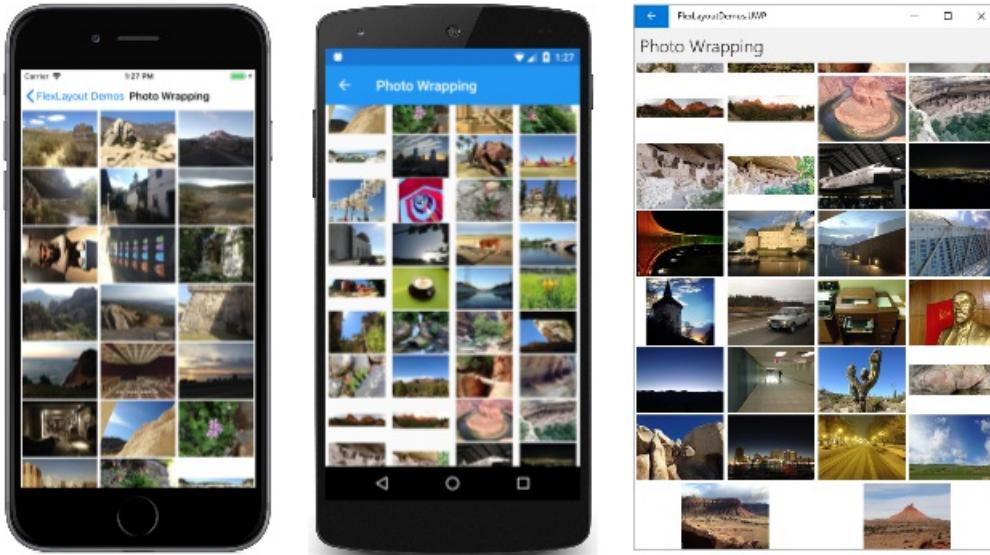
                // Convert to a Stream object
                using (Stream stream = new MemoryStream(data))
                {
                    // Deserialize the JSON into an ImageList object
                    var jsonSerializer = new DataContractJsonSerializer(typeof(ImageList));
                    ImageList imageList = (ImageList)jsonSerializer.ReadObject(stream);

                    // Create an Image object for each bitmap
                    foreach (string filepath in imageList.Photos)
                    {
                        Image image = new Image
                        {
                            Source = ImageSource.FromUri(new Uri(filepath + urlSuffix))
                        };
                        flexLayout.Children.Add(image);
                    }
                }
            }
            catch
            {
                flexLayout.Children.Add(new Label
                {
                    Text = "Cannot access list of bitmap files"
                });
            }
        }

        activityIndicator.IsRunning = false;
        activityIndicator.IsVisible = false;
    }
}

```

Here's the program running, progressively scrolled from top to bottom:



## Page layout with FlexLayout

There is a standard layout in web design called the [holy grail](#) because it's a layout format that is very desirable, but often hard to realize with perfection. The layout consists of a header at the top of the page and a footer at the bottom, both extending to the full width of the page. Occupying the center of the page is the main content, but often with a columnar menu to the left of the content and supplementary information (sometimes called an *aside* area) at the right. [Section 5.4.1 of the CSS Flexible Box Layout specification](#) describes how the holy grail layout can be realized with a flex box.

The **Holy Grail Layout** page of the [FlexLayoutDemos](#) sample shows a simple implementation of this layout using one `FlexLayout` nested in another. Because this page is designed for a phone in portrait mode, the areas to the left and right of the content area are only 50 pixels wide:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlexLayoutDemos.HolyGrailLayoutPage"
    Title="Holy Grail Layout">

    <FlexLayout Direction="Column">

        <!-- Header -->
        <Label Text="HEADER"
            FontSize="Large"
            BackgroundColor="Aqua"
            HorizontalTextAlignment="Center" />

        <!-- Body -->
        <FlexLayout FlexLayout.Grow="1">

            <!-- Content -->
            <Label Text="CONTENT"
                FontSize="Large"
                BackgroundColor="Gray"
                HorizontalTextAlignment="Center"
                VerticalTextAlignment="Center"
                FlexLayout.Grow="1" />

            <!-- Navigation items-->
            <BoxView FlexLayout.Basis="50"
                FlexLayout.Order="-1"
                Color="Blue" />

            <!-- Aside items -->
            <BoxView FlexLayout.Basis="50"
                Color="Green" />

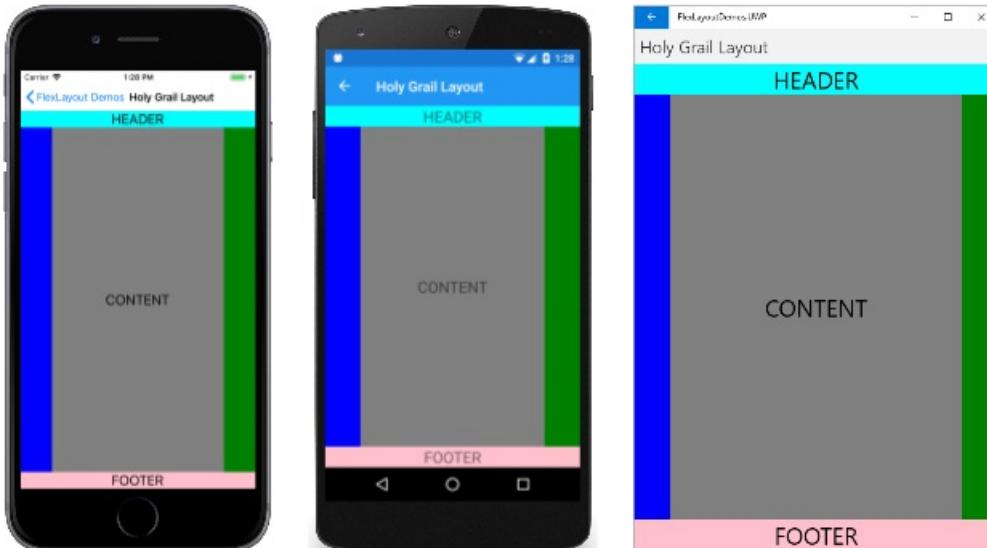
        </FlexLayout>

        <!-- Footer -->
        <Label Text="FOOTER"
            FontSize="Large"
            BackgroundColor="Pink"
            HorizontalTextAlignment="Center" />

    </FlexLayout>
</ContentPage>

```

Here it is running:



The navigation and aside areas are rendered with a `BoxView` on the left and right.

The first `FlexLayout` in the XAML file has a vertical main axis and contains three children arranged in a column. These are the header, the body of the page, and the footer. The nested `FlexLayout` has a horizontal main axis with three children arranged in a row.

Three attached bindable properties are demonstrated in this program:

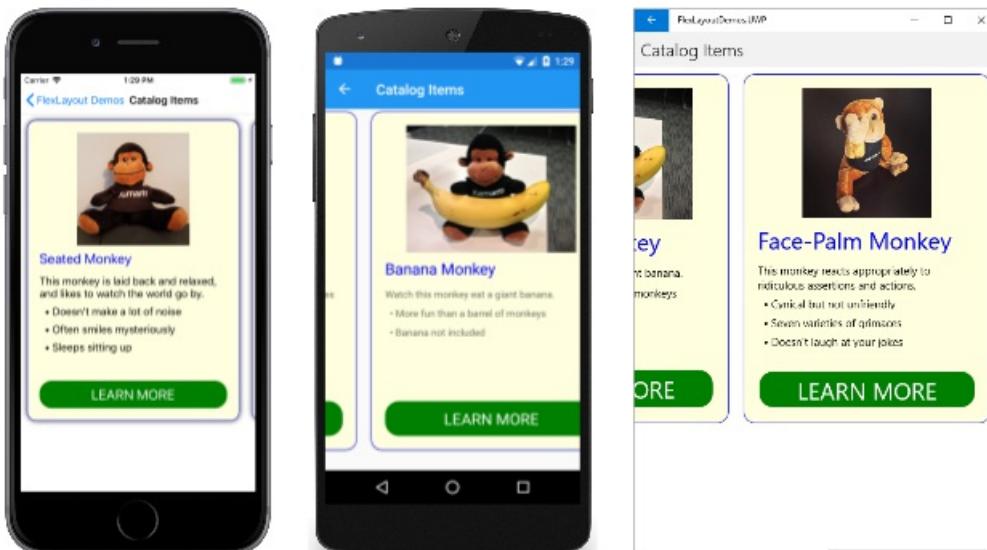
- The `Order` attached bindable property is set on the first `BoxView`. This property is an integer with a default value of 0. You can use this property to change the layout order. Generally developers prefer the content of the page to appear in markup prior to the navigation items and aside items. Setting the `Order` property on the first `BoxView` to a value less than its other siblings causes it to appear as the first item in the row. Similarly, you can ensure that an item appears last by setting the `Order` property to a value greater than its siblings.
- The `Basis` attached bindable property is set on the two `BoxView` items to give them a width of 50 pixels. This property is of type `FlexBasis`, a structure that defines a static property of type `FlexBasis` named `Auto`, which is the default. You can use `Basis` to specify a pixel size or a percentage that indicates how much space the item occupies on the main axis. It is called a *basis* because it specifies an item size that is the basis of all subsequent layout.
- The `Grow` property is set on the nested `Layout` and on the `Label` child representing the content. This property is of type `float` and has a default value of 0. When set to a positive value, all the remaining space along the main axis is allocated to that item and to siblings with positive values of `Grow`. The space is allocated proportionally to the values, somewhat like the star specification in a `Grid`.

The first `Grow` attached property is set on the nested `FlexLayout`, indicating that this `FlexLayout` is to occupy all the unused vertical space within the outer `FlexLayout`. The second `Grow` attached property is set on the `Label` representing the content, indicating that this content is to occupy all the unused horizontal space within the inner `FlexLayout`.

There is also a similar `Shrink` attached bindable property that you can use when the size of children exceeds the size of the `FlexLayout` but wrapping is not desired.

## Catalog items with FlexLayout

The **Catalog Items** page in the [FlexLayoutDemos](#) sample is similar to [Example 1 in Section 1.1 of the CSS Flex Layout Box specification](#) except that it displays a horizontally scrollable series of pictures and descriptions of three monkeys:



Each of the three monkeys is a `FlexLayout` contained in a `Frame` that is given an explicit height and width, and which is also a child of a larger `FlexLayout`. In this XAML file, most of the properties of the `FlexLayout` children

are specified in styles, all but one of which is an implicit style:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:FlexLayoutDemos"
    x:Class="FlexLayoutDemos.CatalogItemsPage"
    Title="Catalog Items">
<ContentPage.Resources>
    <Style TargetType="Frame">
        <Setter Property="BackgroundColor" Value="LightYellow" />
        <Setter Property="BorderColor" Value="Blue" />
        <Setter Property="Margin" Value="10" />
        <Setter Property="CornerRadius" Value="15" />
    </Style>

    <Style TargetType="Label">
        <Setter Property="Margin" Value="0, 4" />
    </Style>

    <Style x:Key="headerLabel" TargetType="Label">
        <Setter Property="Margin" Value="0, 8" />
        <Setter Property="FontSize" Value="Large" />
        <Setter Property="TextColor" Value="Blue" />
    </Style>

    <Style TargetType="Image">
        <Setter Property="FlexLayout.Order" Value="-1" />
        <Setter Property="FlexLayout.AlignSelf" Value="Center" />
    </Style>

    <Style TargetType="Button">
        <Setter Property="Text" Value="LEARN MORE" />
        <Setter Property="FontSize" Value="Large" />
        <Setter Property="TextColor" Value="White" />
        <Setter Property="BackgroundColor" Value="Green" />
        <Setter Property="BorderRadius" Value="20" />
    </Style>
</ContentPage.Resources>

<ScrollView Orientation="Both">
    <FlexLayout>
        <Frame WidthRequest="300"
            HeightRequest="480">

            <FlexLayout Direction="Column">
                <Label Text="Seated Monkey"
                    Style="{StaticResource headerLabel}" />
                <Label Text="This monkey is laid back and relaxed, and likes to watch the world go by." />
                <Label Text="  Doesn't make a lot of noise" />
                <Label Text="  Often smiles mysteriously" />
                <Label Text="  Sleeps sitting up" />
                <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}"
                    WidthRequest="180"
                    HeightRequest="180" />
                <Label FlexLayout.Grow="1" />
                <Button />
            </FlexLayout>
        </Frame>

        <Frame WidthRequest="300"
            HeightRequest="480">

            <FlexLayout Direction="Column">
                <Label Text="Banana Monkey"
                    Style="{StaticResource headerLabel}" />
                <Label Text="Watch this monkey eat a giant banana." />
                <Label Text="  More fun than a barrel of monkeys" />
                <Label Text="  Banana not included" />
            </FlexLayout>
        </Frame>
    </FlexLayout>
</ScrollView>
```

```

<Image Source="{local:ImageResource FlexLayoutDemos.Images.Banana.jpg}"
       WidthRequest="240"
       HeightRequest="180" />
<Label FlexLayout.Grow="1" />
<Button />
</FlexLayout>
</Frame>

<Frame WidthRequest="300"
      HeightRequest="480">

    <FlexLayout Direction="Column">
        <Label Text="Face-Palm Monkey"
              Style="{StaticResource headerLabel}" />
        <Label Text="This monkey reacts appropriately to ridiculous assertions and actions." />
        <Label Text=" &#x2022; Cynical but not unfriendly" />
        <Label Text=" &#x2022; Seven varieties of grimaces" />
        <Label Text=" &#x2022; Doesn't laugh at your jokes" />
        <Image Source="{local:ImageResource FlexLayoutDemos.Images.FacePalm.jpg}"
               WidthRequest="180"
               HeightRequest="180" />
        <Label FlexLayout.Grow="1" />
        <Button />
    </FlexLayout>
</Frame>
</FlexLayout>
</ScrollView>
</ContentPage>

```

The implicit style for the `Image` includes settings of two attached bindable properties of `FlexLayout`:

```

<Style TargetType="Image">
    <Setter Property="FlexLayout.Order" Value="-1" />
    <Setter Property="FlexLayout.AlignSelf" Value="Center" />
</Style>

```

The `Order` setting of `-1` causes the `Image` element to be displayed first in each of the nested `FlexLayout` views regardless of its position within the children collection. The `AlignSelf` property of `Center` causes the `Image` to be centered within the `FlexLayout`. This overrides the setting of the `AlignItems` property, which has a default value of `Stretch`, meaning that the `Label` and `Button` children are stretched to the full width of the `FlexLayout`.

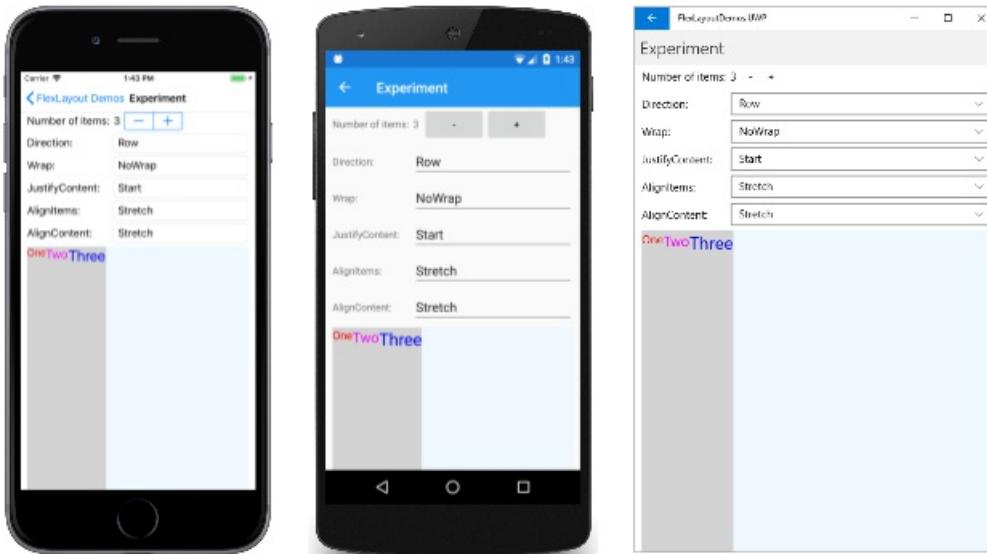
Within each of the three `FlexLayout` views, a blank `Label` precedes the `Button`, but it has a `Grow` setting of `1`. This means that all the extra vertical space is allocated to this blank `Label`, which effectively pushes the `Button` to the bottom.

## The bindable properties in detail

Now that you've seen some common applications of `FlexLayout`, the properties of `FlexLayout` can be explored in more detail. `FlexLayout` defines six bindable properties that you set on the `FlexLayout` itself, either in code or XAML, to control orientation and alignment. (One of these properties, `Position`, is not covered in this article.)

You can experiment with the five remaining bindable properties using the [Experiment](#) page of the [FlexLayoutDemos](#) sample. This page allows you to add or remove children from a `FlexLayout` and to set combinations of the five bindable properties. All the children of the `FlexLayout` are `Label` views of various colors and sizes, with the `Text` property set to a number corresponding to its position in the `Children` collection.

When the program starts up, five `Picker` views display the default values of these five `FlexLayout` properties. The `FlexLayout` towards the bottom of the screen contains three children:



Each of the `Label` views has a gray background that shows the space allocated to that `Label` within the `FlexLayout`. The background of the `FlexLayout` itself is Alice Blue. It occupies the entire bottom area of the page except for a little margin at the left and right.

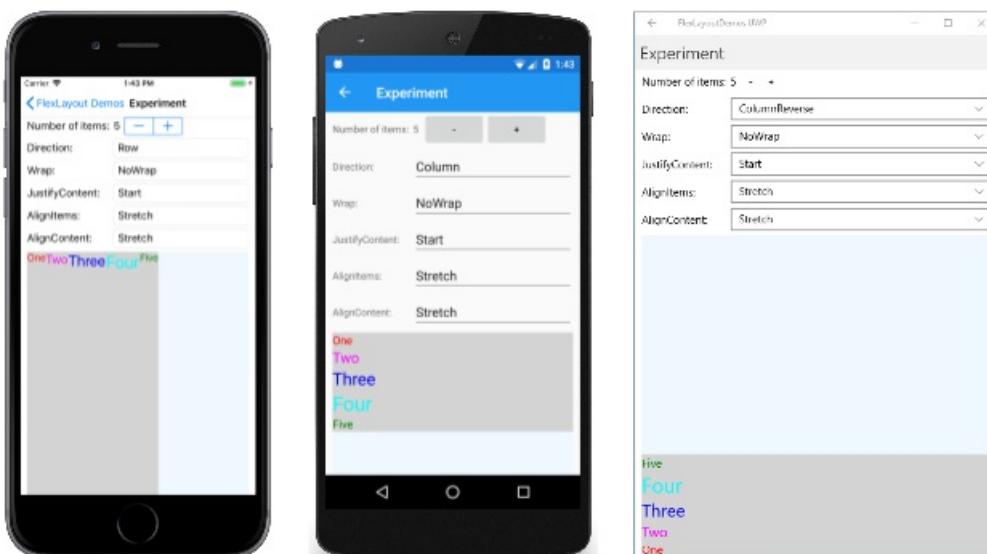
### The Direction property

The `Direction` property is of type `FlexDirection`, an enumeration with four members:

- `Column`
- `ColumnReverse` (or "column-reverse" in XAML)
- `Row`, the default
- `RowReverse` (or "row-reverse" in XAML)

In XAML, you can specify the value of this property using the enumeration member names in lowercase, uppercase, or mixed case, or you can use two additional strings shown in parentheses that are the same as the CSS indicators. (The "column-reverse" and "row-reverse" strings are defined in the `FlexDirectionTypeConverter` class used by the XAML parser.)

Here's the **Experiment** page showing (from left to right) the `Row` direction, `Column` direction, and `ColumnReverse` direction:



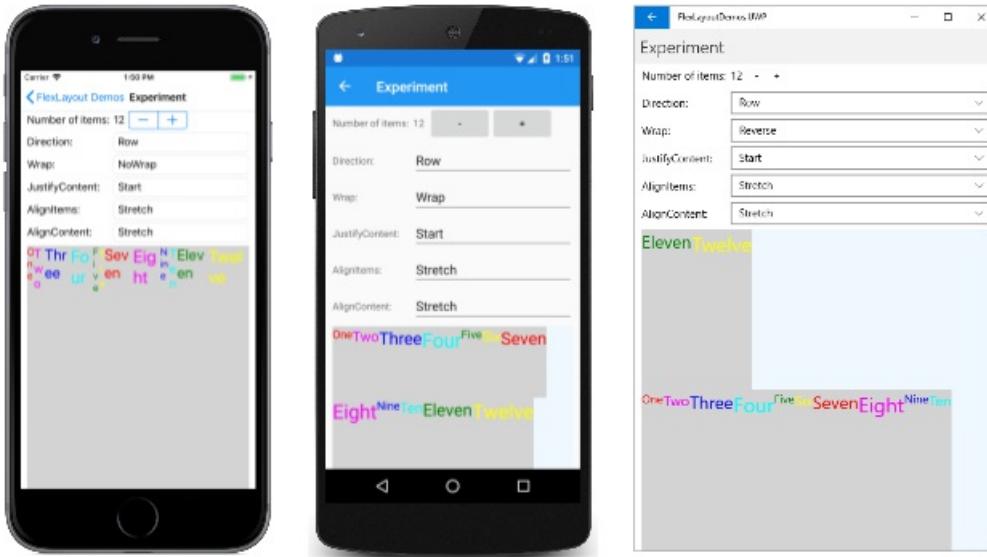
Notice that for the `Reverse` options, the items start at the right or bottom.

### The Wrap property

The `Wrap` property is of type `FlexWrap`, an enumeration with three members:

- `NoWrap`, the default
- `Wrap`
- `Reverse` (or "wrap-reverse" in XAML)

From left to right, these screens show the `NoWrap`, `Wrap` and `Reverse` options for 12 children:



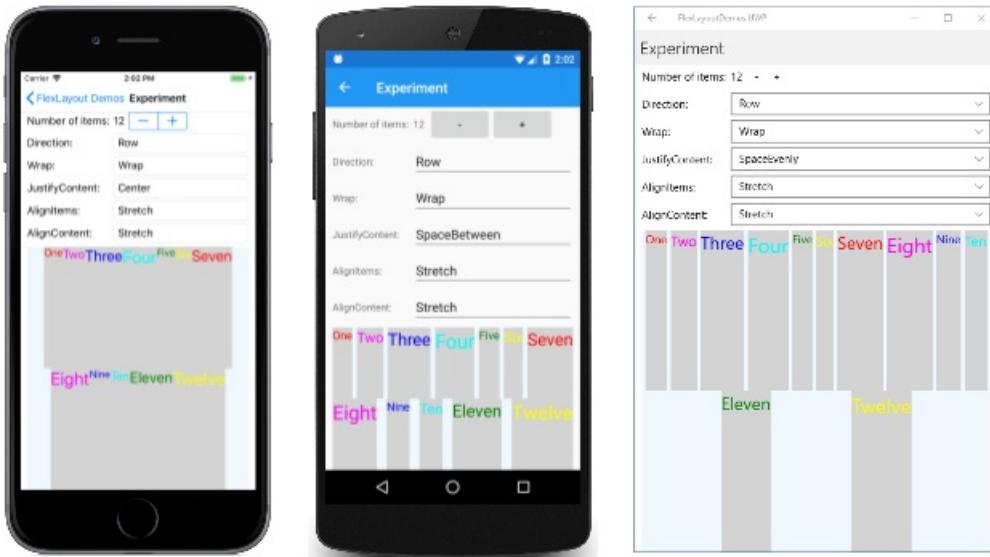
When the `Wrap` property is set to `NoWrap` and the main axis is constrained (as in this program), and the main axis is not wide or tall enough to fit all the children, the `FlexLayout` attempts to make the items smaller, as the iOS screenshot demonstrates. You can control the shrinkage of the items with the `Shrink` attached bindable property.

### The JustifyContent property

The `JustifyContent` property is of type `FlexJustify`, an enumeration with six members:

- `Start` (or "flex-start" in XAML), the default
- `Center`
- `End` (or "flex-end" in XAML)
- `SpaceBetween` (or "space-between" in XAML)
- `SpaceAround` (or "space-around" in XAML)
- `SpaceEvenly`

This property specifies how the items are spaced on the main axis, which is the horizontal axis in this example:



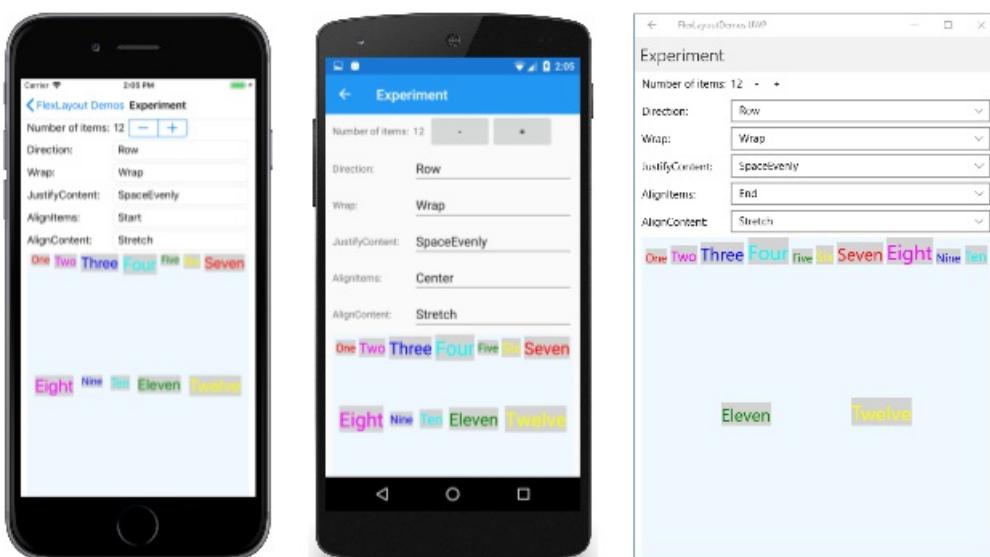
In all three screenshots, the `Wrap` property is set to `Wrap`. The `Start` default is shown in the previous Android screenshot. The iOS screenshot here shows the `Center` option: all the items are moved to the center. The three other options beginning with the word `Space` allocate the extra space not occupied by the items. `SpaceBetween` allocates the space equally between the items; `SpaceAround` puts equal space around each item, while `SpaceEvenly` puts equal space between each item, and before the first item and after the last item on the row.

### The `AlignItems` property

The `AlignItems` property is of type `FlexAlignItems`, an enumeration with four members:

- `Stretch`, the default
- `Center`
- `Start` (or "flex-start" in XAML)
- `End` (or "flex-end" in XAML)

This is one of two properties (the other being `AlignContent`) that indicates how children are aligned on the cross axis. Within each row, the children are stretched (as shown in the previous screenshot), or aligned on the start, center, or end of each item, as shown in the following three screenshots:



In the iOS screenshot, the tops of all the children are aligned. In the Android screenshots, the items are vertically centered based on the tallest child. In the UWP screenshot, the bottoms of all the items are aligned.

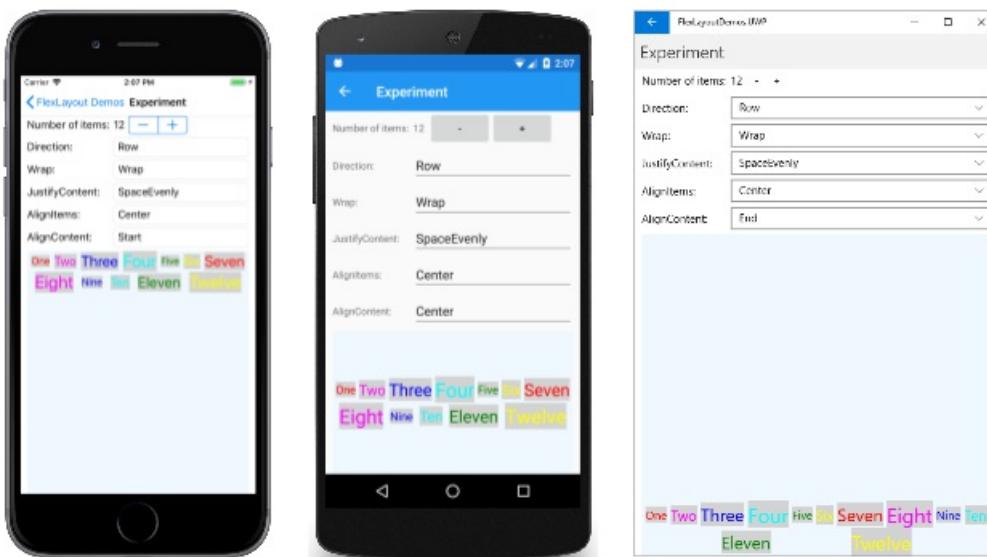
For any individual item, the `AlignItems` setting can be overridden with the `AlignSelf` attached bindable property.

## The AlignContent property

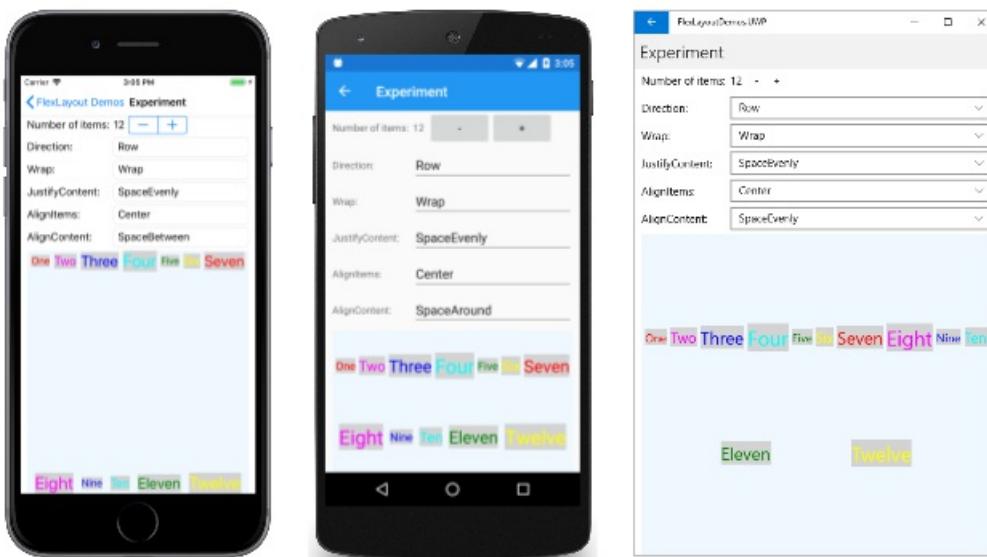
The `AlignContent` property is of type `FlexAlignContent`, an enumeration with seven members:

- `Stretch`, the default
- `Center`
- `Start` (or "flex-start" in XAML)
- `End` (or "flex-end" in XAML)
- `SpaceBetween` (or "space-between" in XAML)
- `SpaceAround` (or "space-around" in XAML)
- `SpaceEvenly`

Like `AlignItems`, the `AlignContent` property also aligns children on the cross axis, but affects entire rows or columns:



In the iOS screenshot, both rows are at the top; in the Android screenshot they're in the center; and in the UWP screenshot they're at the bottom. The rows can also be spaced in various ways:



The `AlignContent` has no effect when there is only one row or column.

## The attached bindable properties in detail

`FlexLayout` defines five attached bindable properties. These properties are set on children of the `FlexLayout` and

pertain only to that particular child.

## The AlignSelf Property

The `AlignSelf` attached bindable property is of type `FlexAlignSelf`, an enumeration with five members:

- `Auto`, the default
- `Stretch`
- `Center`
- `Start` (or "flex-start" in XAML)
- `End` (or "flex-end" in XAML)

For any individual child of the `FlexLayout`, this property setting overrides the `AlignItems` property set on the `FlexLayout` itself. The default setting of `Auto` means to use the `AlignItems` setting.

For a `Label` element named `label` (or example), you can set the `AlignSelf` property in code like this:

```
FlexAlign.SetAlignSelf(label, FlexAlignSelf.Center);
```

Notice that there is no reference to the `FlexLayout` parent of the `Label`. In XAML, you set the property like this:

```
<Label ... FlexAlign.AlignSelf="Center" ... />
```

## The Order Property

The `Order` property is of type `int`. The default value is 0.

The `Order` property allows you to change the order that the children of the `FlexLayout` are arranged. Usually, the children of a `FlexLayout` are arranged in the same order that they appear in the `Children` collection. You can override this order by setting the `Order` attached bindable property to a non-zero integer value on one or more children. The `FlexLayout` then arranges its children based on the setting of the `Order` property on each child, but children with the same `Order` setting are arranged in the order that they appear in the `Children` collection.

## The Basis Property

The `Basis` attached bindable property indicates the amount of space that is allocated to a child of the `FlexLayout` on the main axis. The size specified by the `Basis` property is the size along the main axis of the parent `FlexLayout`. Therefore, `Basis` indicates the width of a child when the children are arranged in rows, or the height when the children are arranged in columns.

The `Basis` property is of type `FlexBasis`, a structure. The size can be specified in either device-independent units or as a percentage of the size of the `FlexLayout`. The default value of the `Basis` property is the static property `FlexBasis.Auto`, which means that the child's requested width or height is used.

In code, you can set the `Basis` property for a `Label` named `label` to 40 device-independent units like this:

```
FlexLayout.SetBasis(label, new FlexBasis(40, false));
```

The second argument to the `FlexBasis` constructor is named `isRelative` and indicates whether the size is relative (`true`) or absolute (`false`). The argument has a default value of `false`, so you can also use the following code:

```
FlexLayout.SetBasis(label, new FlexBasis(40));
```

An implicit conversion from `float` to `FlexBasis` is defined, so you can simplify it even further:

```
FlexLayout.SetBasis(label, 40);
```

You can set the size to 25% of the `FlexLayout` parent like this:

```
FlexLayout.SetBasis(label, new FlexBasis(0.25f, true));
```

This fractional value must be in the range of 0 to 1.

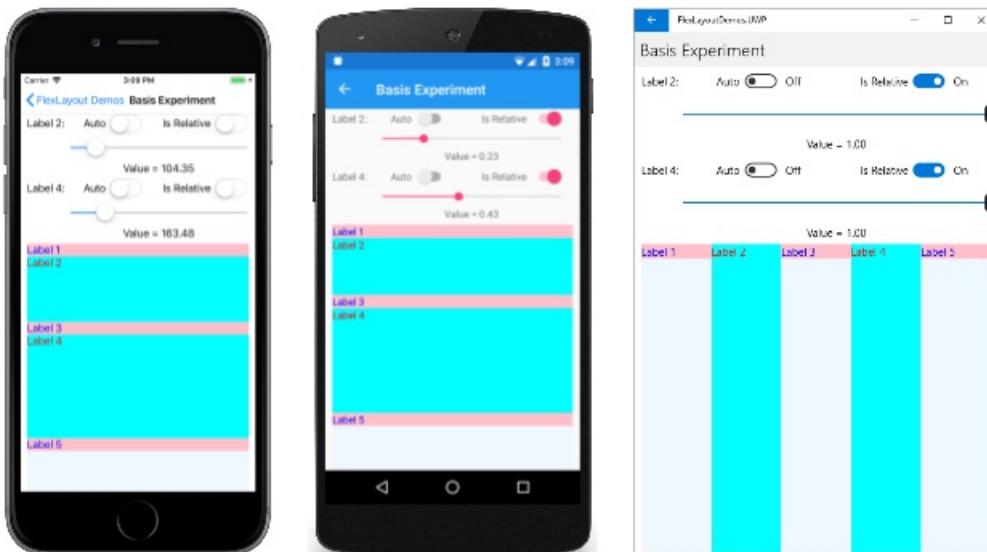
In XAML, you can use a number for a size in device-independent units:

```
<Label ... FlexLayout.Basis="40" ... />
```

Or you can specify a percentage in the range of 0% to 100%:

```
<Label ... FlexLayout.Basis="25%" ... />
```

The **Basis Experiment** page of the [FlexLayoutDemos](#) sample allows you to experiment with the `Basis` property. The page displays a wrapped column of five `Label` elements with alternating background and foreground colors. Two `Slider` elements let you specify `Basis` values for the second and fourth `Label`:



The iOS screenshot at the left shows the two `Label` elements being given heights in device-independent units. The Android screen shows them being given heights that are a fraction of the total height of the `FlexLayout`. If the `Basis` is set at 100%, then the child is the height of the `FlexLayout`, and will wrap to the next column and occupy the entire height of that column, as the UWP screenshot demonstrates: It appears as if the five children are arranged in a row, but they're actually arranged in five columns.

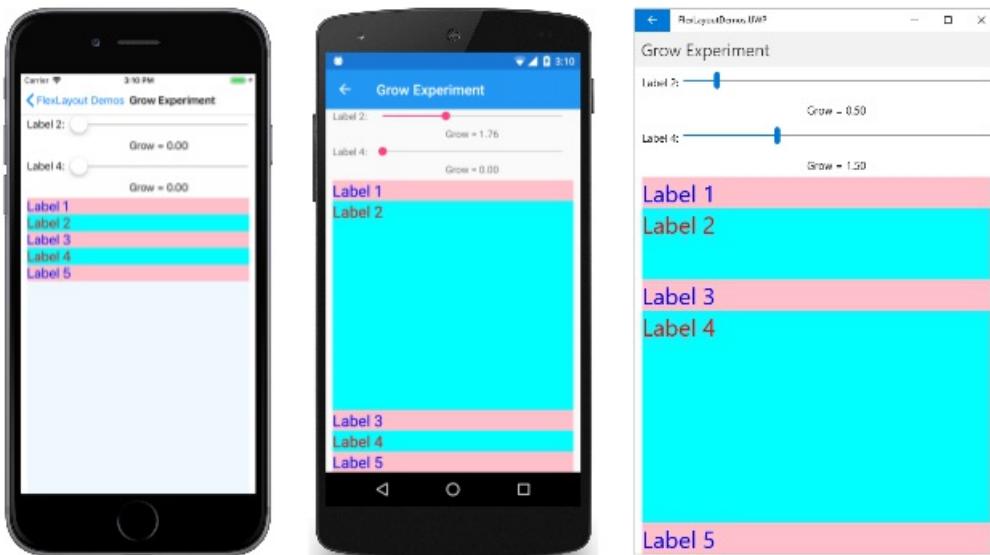
### The Grow Property

The `Grow` attached bindable property is of type `int`. The default value is 0, and the value must be greater than or equal to 0.

The `Grow` property plays a role when the `Wrap` property is set to `NoWrap` and the row of children has a total width less than the width of the `FlexLayout`, or the column of children has a shorter height than the `FlexLayout`. The `Grow` property indicates how to apportion the leftover space among the children.

In the **Grow Experiment** page, five `Label` elements of alternating colors are arranged in a column, and two `Slider` elements allow you to adjust the `Grow` property of the second and fourth `Label`. The iOS screenshot at

the far left shows the default `Grow` properties of 0:



If any one child is given a positive `Grow` value, then that child takes up all the remaining space, as the Android screenshot demonstrates. This space can also be allocated among two or more children. In the UWP screenshot, the `Grow` property of the second `Label` is set to 0.5, while the `Grow` property of the fourth `Label` is 1.5, which gives the fourth `Label` three times as much of the leftover space as the second `Label`.

How the child view uses that space depends on the particular type of child. For a `Label`, the text can be positioned within the total space of the `Label` using the properties `HorizontalTextAlignment` and `VerticalTextAlignment`.

### The Shrink Property

The `Shrink` attached bindable property is of type `int`. The default value is 1, and the value must be greater than or equal to 0.

The `Shrink` property plays a role when the `Wrap` property is set to `NoWrap` and the aggregate width of a row of children is greater than the width of the `FlexLayout`, or the aggregate height of a single column of children is greater than the height of the `FlexLayout`. Normally the `FlexLayout` will display these children by constricting their sizes. The `shrink` property can indicate which children are given priority in being displayed at their full sizes.

The **Shrink Experiment** page creates a `FlexLayout` with a single row of five `Label` children that require more space than the `FlexLayout` width. The iOS screenshot at the left shows all the `Label` elements with default values of 1:



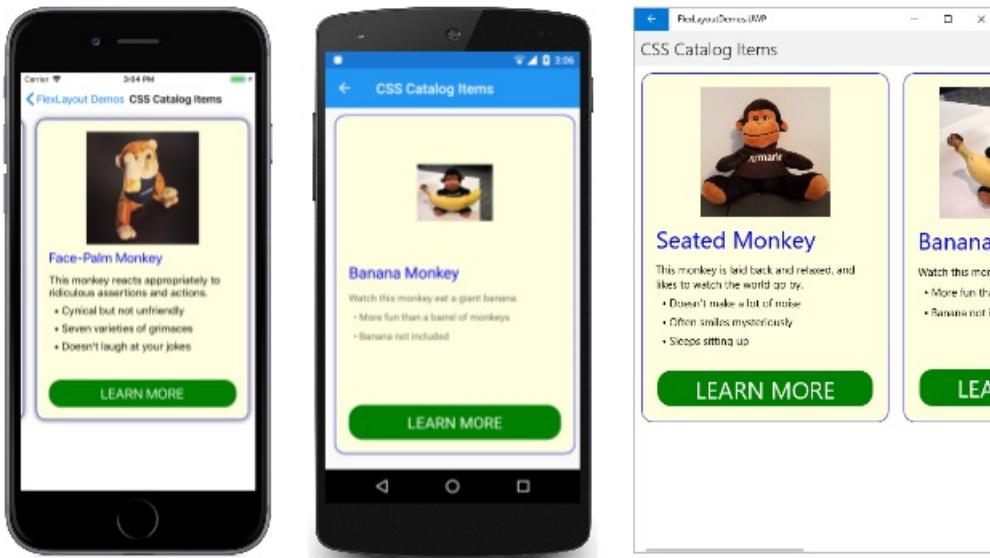
In the Android screenshot, the `Shrink` value for the second `Label` is set to 0, and that `Label` is displayed in its

full width. Also, the fourth `Label` is given a `Shrink` value greater than one, and it has shrunk. The UWP screenshot shows both `Label` elements being given a `Shrink` value of 0 to allow them to be displayed in their full size, if that is possible.

You can set both the `Grow` and `Shrink` values to accommodate situations where the aggregate child sizes might sometimes be less than or sometimes greater than the size of the `FlexLayout`.

## CSS styling with FlexLayout

You can use the [CSS styling](#) feature introduced with Xamarin.Forms 3.0 in connection with `FlexLayout`. The [CSS Catalog Items](#) page of the [FlexLayoutDemos](#) sample duplicates the layout of the [Catalog Items](#) page, but with a CSS style sheet for many of the styles:



The original `CatalogItemsPage.xaml` file has five `Style` definitions in its `Resources` section with 15 `Setter` objects. In the `CssCatalogItemsPage.xaml` file, that has been reduced to two `Style` definitions with just four `Setter` objects. These styles supplement the CSS style sheet for properties that the Xamarin.Forms CSS styling feature currently doesn't support:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:FlexLayoutDemos"
    x:Class="FlexLayoutDemos.CssCatalogItemsPage"
    Title="CSS Catalog Items">
    <ContentPage.Resources>
        <StyleSheet Source="CatalogItemsStyles.css" />

        <Style TargetType="Frame">
            <Setter Property="BorderColor" Value="Blue" />
            <Setter Property="CornerRadius" Value="15" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="Text" Value="LEARN MORE" />
            <Setter Property="BorderRadius" Value="20" />
        </Style>
    </ContentPage.Resources>

    <ScrollView Orientation="Both">
        <FlexLayout>
            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Seated Monkey" StyleClass="header" />
                    <Label Text="This monkey is laid back and relaxed, and likes to watch the world go by." />
                    <Label Text=" &#x2022; Doesn't make a lot of noise" />
                    <Label Text=" &#x2022; Often smiles mysteriously" />
                    <Label Text=" &#x2022; Sleeps sitting up" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Banana Monkey" StyleClass="header" />
                    <Label Text="Watch this monkey eat a giant banana." />
                    <Label Text=" &#x2022; More fun than a barrel of monkeys" />
                    <Label Text=" &#x2022; Banana not included" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.Banana.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Face-Palm Monkey" StyleClass="header" />
                    <Label Text="This monkey reacts appropriately to ridiculous assertions and actions." />
                    <Label Text=" &#x2022; Cynical but not unfriendly" />
                    <Label Text=" &#x2022; Seven varieties of grimaces" />
                    <Label Text=" &#x2022; Doesn't laugh at your jokes" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.FacePalm.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>
        </FlexLayout>
    </ScrollView>
</ContentPage>

```

The CSS style sheet is referenced in the first line of the `Resources` section:

```
<StyleSheet Source="CatalogItemsStyles.css" />
```

Notice also that two elements in each of the three items include `StyleClass` settings:

```
<Label Text="Seated Monkey" StyleClass="header" />
...
<Label StyleClass="empty" />
```

These refer to selectors in the **CatalogItemsStyles.css** style sheet:

```
frame {
    width: 300;
    height: 480;
    background-color: lightyellow;
    margin: 10;
}

label {
    margin: 4 0;
}

label.header {
    margin: 8 0;
    font-size: large;
    color: blue;
}

label.empty {
    flex-grow: 1;
}

image {
    height: 180;
    order: -1;
    align-self: center;
}

button {
    font-size: large;
    color: white;
    background-color: green;
}
```

Several `FlexLayout` attached bindable properties are referenced here. In the `label.empty` selector, you'll see the `flex-grow` attribute, which styles an empty `Label` to provide some blank space above the `Button`. The `image` selector contains an `order` attribute and an `align-self` attribute, both of which correspond to `FlexLayout` attached bindable properties.

You've seen that you can set properties directly on the `FlexLayout` and you can set attached bindable properties on the children of a `FlexLayout`. Or, you can set these properties indirectly using traditional XAML-based styles or CSS styles. What's important is to know and understand these properties. These properties are what makes the `FlexLayout` truly flexible.

## FlexLayout with Xamarin.University

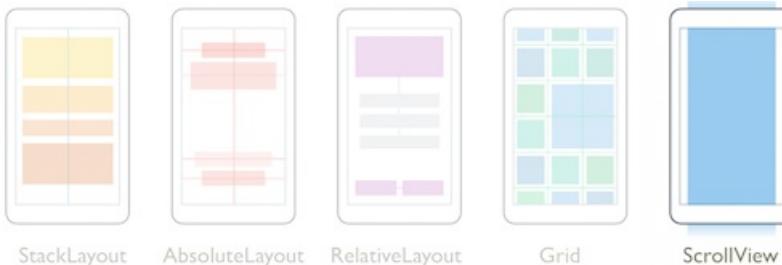
## Related links

- [FlexLayoutDemos](#)

# Xamarin.Forms ScrollView

7/31/2018 • 3 minutes to read • [Edit Online](#)

`ScrollView` contains layouts and enables them to scroll offscreen. `ScrollView` is also used to allow views to automatically move to the visible portion of the screen when the keyboard is showing.



This article covers:

- **Purpose** – the purpose for `ScrollView` and when it is used.
- **Usage** – how to use `ScrollView` in practice.
- **Properties** – public properties that can be read and modified.
- **Methods** – public methods that can be called to scroll the view.
- **Events** – events that can be used to listen to changes in the view's states.

## Purpose

`ScrollView` can be used to ensure that larger views display well on smaller phones. For example, a layout that works on an iPhone 6s may be clipped on an iPhone 4s. Using a `ScrollView` would allow the clipped portions of the layout to be displayed on the smaller screen.

## Usage

### NOTE

`ScrollView`s should not be nested. In addition, `ScrollView`s should not be nested with other controls that provide scrolling, like `ListView` and `WebView`.

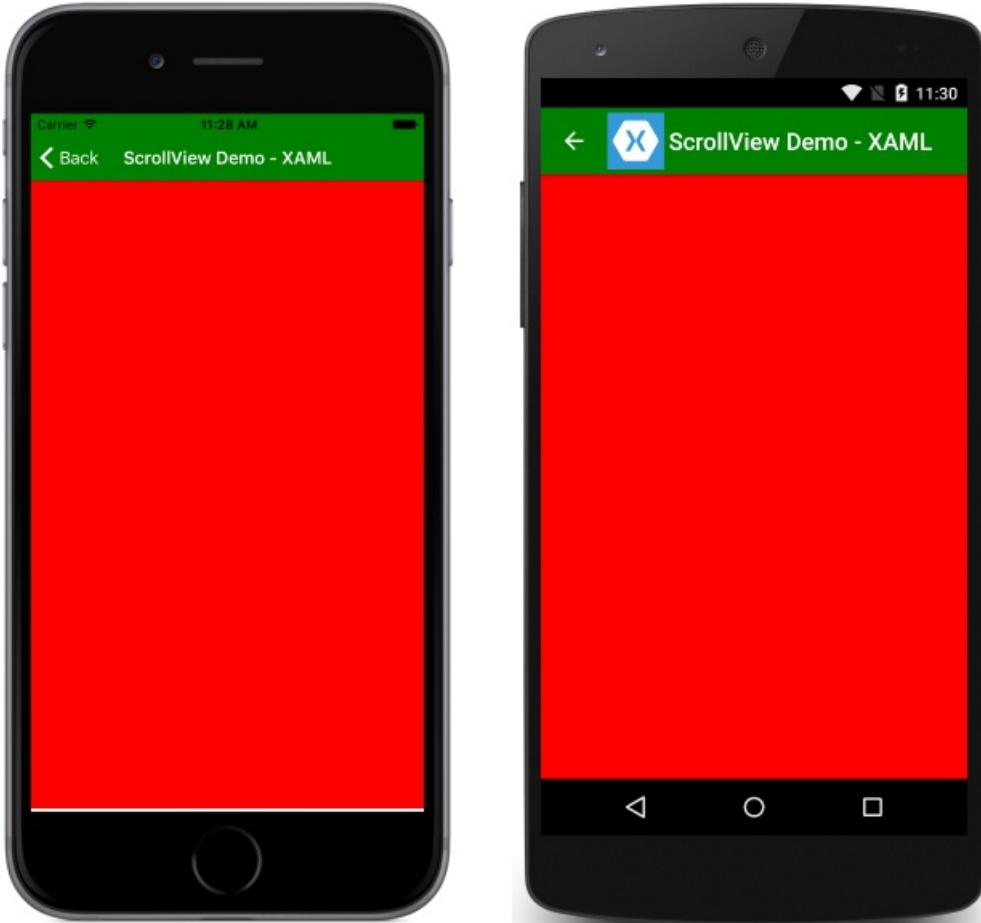
`ScrollView` exposes a `Content` property which can be set to a single view or layout. Consider this example of a layout with a very large boxView, followed by an `Entry`:

```
<ContentPage.Content>
    <ScrollView>
        <StackLayout>
            <BoxView BackgroundColor="Red" HeightRequest="600" WidthRequest="150" />
            <Entry />
        </StackLayout>
    </ScrollView>
</ContentPage.Content>
```

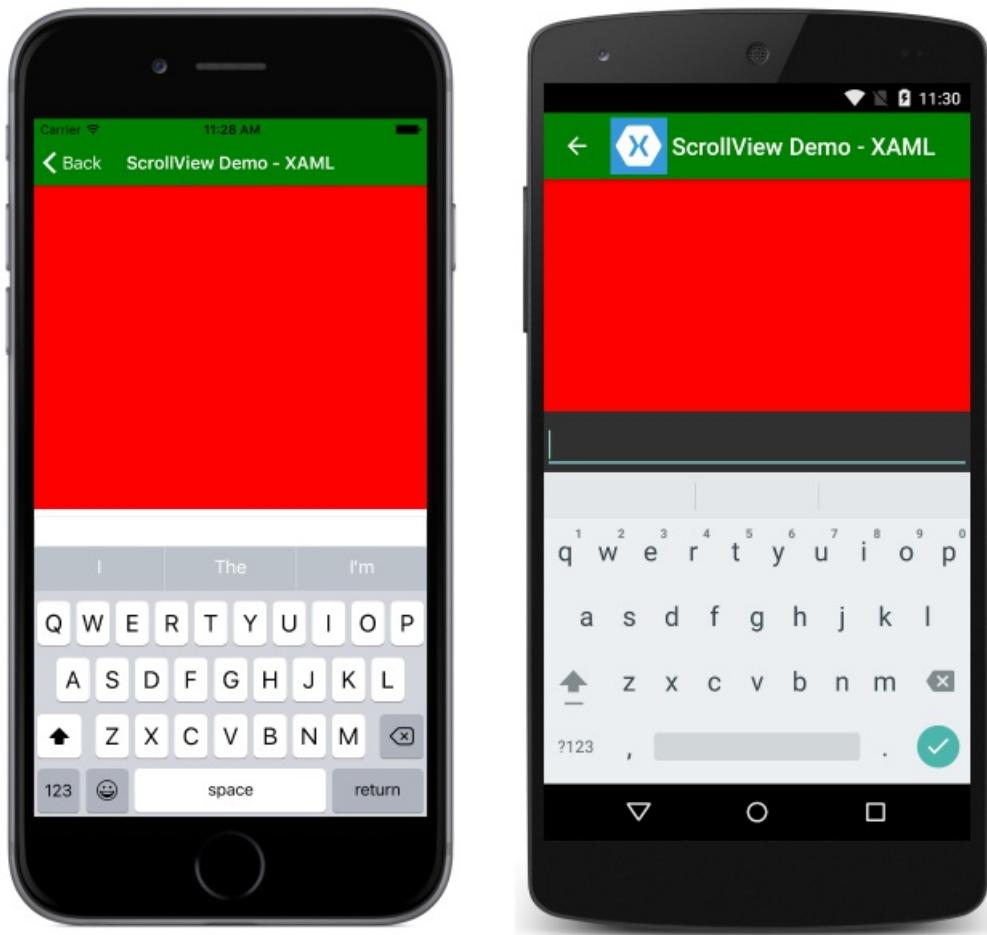
In C#:

```
var scroll = new ScrollView();
Content = scroll;
var stack = new StackLayout();
stack.Children.Add(new BoxView { BackgroundColor = Color.Red, HeightRequest = 600, WidthRequest = 600 });
stack.Children.Add(new Entry());
```

Before the user scrolls down, only the `BoxView` is visible:



Notice that when the user starts to enter text in the `Entry`, the view scrolls to keep it visible on screen:



## Properties

`ScrollView` defines the following properties:

- `ContentSize` gets a `Size` value that represents the size of the content.
- `Orientation` gets or sets a `ScrollOrientation` enumeration value that represents the scrolling direction of the `ScrollView`.
- `ScrollX` gets a `double` that represents the current X scroll position.
- `ScrollY` gets a `double` that represents the current Y scroll position.
- `HorizontalScrollBarVisibility` gets or sets a `ScrollBarVisibility` value that represents when the horizontal scroll bar is visible.
- `VerticalScrollBarVisibility` gets or sets a `ScrollBarVisibility` value that represents when the vertical scroll bar is visible.

## Methods

`ScrollView` provides a `ScrollToAsync` method, which can be used to scroll the view either using coordinates or by specifying a particular view that should be made visible.

When using coordinates, specify the `x` and `y` coordinates, along with a boolean indicating whether the scrolling should be animated:

```
scroll.ScrollToAsync(0, 150, true); //scrolls so that the position at 150px from the top is visible  
scroll.ScrollToAsync(label, ScrollToPosition.Start, true); //scrolls so that the label is at the start of the list
```

When scrolling to a particular element, the `ScrollToPosition` enumeration specifies where in the view the element will appear:

- **Center** – scrolls the element to the center of the visible portion of the view.
- **End** – scrolls the element to the end of the visible portion of the view.
- **MakeVisible** – scrolls the element so that it is visible within the view.
- **Start** – scrolls the element to the start of the visible portion of the view.

The `IsAnimated` property specifies how the view will be scrolled. When set to true, a smooth animation will be used, rather than instantly moving the content into view.

## Events

`ScrollView` defines just one event, `Scrolled`. `Scrolled` is raised when the view has finished scrolling. The event handler for `Scrolled` takes `ScrolledEventArgs`, which has the `ScrollX` and `ScrollY` properties. The following demonstrates how to update a label with the current scroll position of a `ScrollView`:

```
Label label = new Label { Text = "Position: " };
ScrollView scroll = new ScrollView();
scroll.Scrolled += (object sender, ScrolledEventArgs e) => {
    label.Text = "Position: " + e.ScrollX + " x " + e.ScrollY;
};
```

Note that scroll positions may be negative, due to the bounce effect when scrolling at the end of a list.

## Related Links

- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)

# Layout Options in Xamarin.Forms

7/12/2018 • 5 minutes to read • [Edit Online](#)

Every `Xamarin.Forms` view has `HorizontalOptions` and `VerticalOptions` properties, of type `LayoutOptions`. This article explains the effect that each `LayoutOptions` value has on the alignment and expansion of a view.

## Overview

The `LayoutOptions` structure encapsulates two layout preferences:

- **Alignment** – the view's preferred alignment, which determines its position and size within its parent layout.
- **Expansion** – used only by a `StackLayout`, and indicates if the view should use extra space, if it's available.

These layout preferences can be applied to a `View`, relative to its parent, by setting the `HorizontalOptions` or `VerticalOptions` property of the `View` to one of the public fields from the `LayoutOptions` structure. The public fields are as follows:

- `Start`
- `Center`
- `End`
- `Fill`
- `StartAndExpand`
- `CenterAndExpand`
- `EndAndExpand`
- `FillAndExpand`

The `Start`, `Center`, `End`, and `Fill` fields are used to define the view's alignment within the parent layout:

- For horizontal alignment, `Start` positions the `View` on the left hand side of the parent layout, and for vertical alignment, it positions the `View` at the top of the parent layout.
- For horizontal and vertical alignment, `Center` horizontally or vertically centers the `View`.
- For horizontal alignment, `End` positions the `View` on the right hand side of the parent layout, and for vertical alignment, it positions the `View` at the bottom of the parent layout.
- For horizontal alignment, `Fill` ensures that the `View` fills the width of the parent layout, and for vertical alignment, it ensures that the `View` fills the height of the parent layout.

The `StartAndExpand`, `CenterAndExpand`, `EndAndExpand`, and `FillAndExpand` values are used to define the alignment preference, and whether the view will occupy more space if available within the parent `StackLayout`.

### NOTE

The default value of a view's `HorizontalOptions` and `VerticalOptions` properties is `LayoutOptions.Fill`.

## Alignment

Alignment controls how a view is positioned within its parent layout when the parent layout contains unused space (that is, the parent layout is larger than the combined size of all its children).

A `StackLayout` only respects the `Start`, `Center`, `End`, and `Fill` `LayoutOptions` fields on child views that are in

the opposite direction to the `StackLayout` orientation. Therefore, child views within a vertically oriented `StackLayout` can set their `HorizontalOptions` properties to one of the `Start`, `Center`, `End`, or `Fill` fields. Similarly, child views within a horizontally oriented `StackLayout` can set their `VerticalOptions` properties to one of the `Start`, `Center`, `End`, or `Fill` fields.

A `StackLayout` does not respect the `Start`, `Center`, `End`, and `Fill` `LayoutOptions` fields on child views that are in the same direction as the `StackLayout` orientation. Therefore, a vertically oriented `StackLayout` ignores the `Start`, `Center`, `End`, or `Fill` fields if they are set on the `VerticalOptions` properties of child views. Similarly, a horizontally oriented `StackLayout` ignores the `Start`, `Center`, `End`, or `Fill` fields if they are set on the `HorizontalOptions` properties of child views.

#### NOTE

`LayoutOptions.Fill` generally overrides size requests specified using the `HeightRequest` and `WidthRequest` properties.

The following XAML code example demonstrates a vertically oriented `StackLayout` where each child `Label` sets its `HorizontalOptions` property to one of the four alignment fields from the `LayoutOptions` structure:

```
<StackLayout Margin="0,20,0,0">
    ...
    <Label Text="Start" BackgroundColor="Gray" HorizontalOptions="Start" />
    <Label Text="Center" BackgroundColor="Gray" HorizontalOptions="Center" />
    <Label Text="End" BackgroundColor="Gray" HorizontalOptions="End" />
    <Label Text="Fill" BackgroundColor="Gray" HorizontalOptions="Fill" />
</StackLayout>
```

The equivalent C# code is shown below:

```
Content = new StackLayout
{
    Margin = new Thickness(0, 20, 0, 0),
    Children = {
        ...
        new Label { Text = "Start", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Start },
        new Label { Text = "Center", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Center },
        new Label { Text = "End", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.End },
        new Label { Text = "Fill", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Fill }
    }
};
```

The code results in the layout shown in the following screenshots:



## Expansion

Expansion controls whether a view will occupy more space, if available, within a `StackLayout`. If the `StackLayout` contains unused space (that is, the `StackLayout` is larger than the combined size of all of its children), the unused space is shared equally by all child views that request expansion by setting their `HorizontalOptions` or `VerticalOptions` properties to a `LayoutOptions` field that uses the `AndExpand` suffix. Note that when all the space in the `StackLayout` is used, the expansion options have no effect.

A `StackLayout` can only expand child views in the direction of its orientation. Therefore, a vertically oriented `StackLayout` can expand child views that set their `VerticalOptions` properties to one of the `StartAndExpand`, `CenterAndExpand`, `EndAndExpand`, or `FillAndExpand` fields, if the `StackLayout` contains unused space. Similarly, a horizontally oriented `StackLayout` can expand child views that set their `HorizontalOptions` properties to one of the `StartAndExpand`, `CenterAndExpand`, `EndAndExpand`, or `FillAndExpand` fields, if the `StackLayout` contains unused space.

A `StackLayout` can't expand child views in the direction opposite to its orientation. Therefore, on a vertically oriented `StackLayout`, setting the `HorizontalOptions` property on a child view to `StartAndExpand` has the same effect as setting the property to `Start`.

### NOTE

Note that enabling expansion doesn't change the size of a view unless it uses `LayoutOptions.FillAndExpand`.

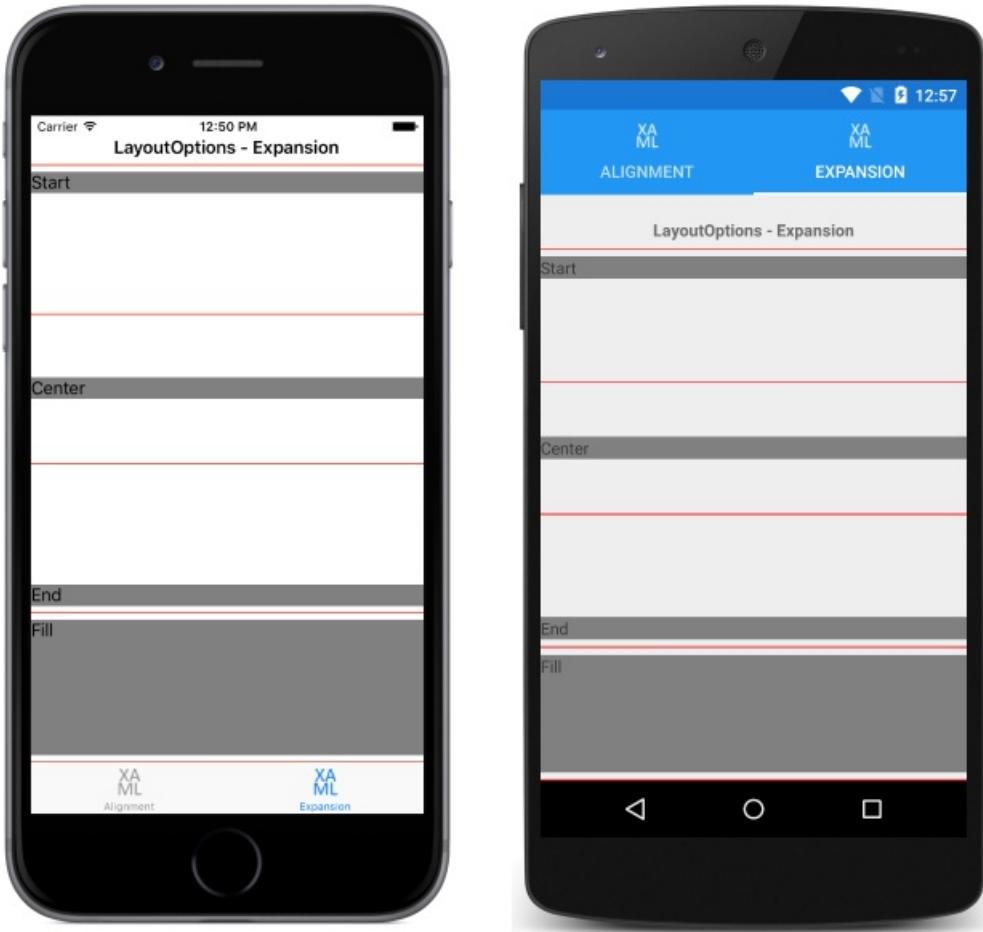
The following XAML code example demonstrates a vertically oriented `StackLayout` where each child `Label` sets its `VerticalOptions` property to one of the four expansion fields from the `LayoutOptions` structure:

```
<StackLayout Margin="0,20,0,0">
    ...
    <BoxView BackgroundColor="Red" HeightRequest="1" />
    <Label Text="Start" BackgroundColor="Gray" VerticalOptions="StartAndExpand" />
    <BoxView BackgroundColor="Red" HeightRequest="1" />
    <Label Text="Center" BackgroundColor="Gray" VerticalOptions="CenterAndExpand" />
    <BoxView BackgroundColor="Red" HeightRequest="1" />
    <Label Text="End" BackgroundColor="Gray" VerticalOptions="EndAndExpand" />
    <BoxView BackgroundColor="Red" HeightRequest="1" />
    <Label Text="Fill" BackgroundColor="Gray" VerticalOptions="FillAndExpand" />
    <BoxView BackgroundColor="Red" HeightRequest="1" />
</StackLayout>
```

The equivalent C# code is shown below:

```
Content = new StackLayout
{
    Margin = new Thickness(0, 20, 0, 0),
    Children = {
        ...
        new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
        new Label { Text = "StartAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.StartAndExpand },
        new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
        new Label { Text = "CenterAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.CenterAndExpand },
        new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
        new Label { Text = "EndAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.EndAndExpand },
        new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
        new Label { Text = "FillAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.FillAndExpand },
        new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 }
    }
};
```

The code results in the layout shown in the following screenshots:



Each `Label` occupies the same amount of space within the `StackLayout`. However, only the final `Label`, which sets its `VerticalOptions` property to `FillAndExpand` has a different size. In addition, each `Label` is separated by a small red `BoxView`, which enables the space the `Label` occupies to be easily viewed.

## Summary

This article explained the effect that each `LayoutOptions` structure value has on the alignment and expansion of a view, relative to its parent. The `Start`, `Center`, `End`, and `Fill` fields are used to define the view's alignment within the parent layout, and the `StartAndExpand`, `CenterAndExpand`, `EndAndExpand`, and `FillAndExpand` fields are used to define the alignment preference, and to determine whether the view will occupy more space, if available, within a `StackLayout`.

## Related Links

- [LayoutOptions \(sample\)](#)
- [LayoutOptions](#)

# Margin and Padding

7/12/2018 • 2 minutes to read • [Edit Online](#)

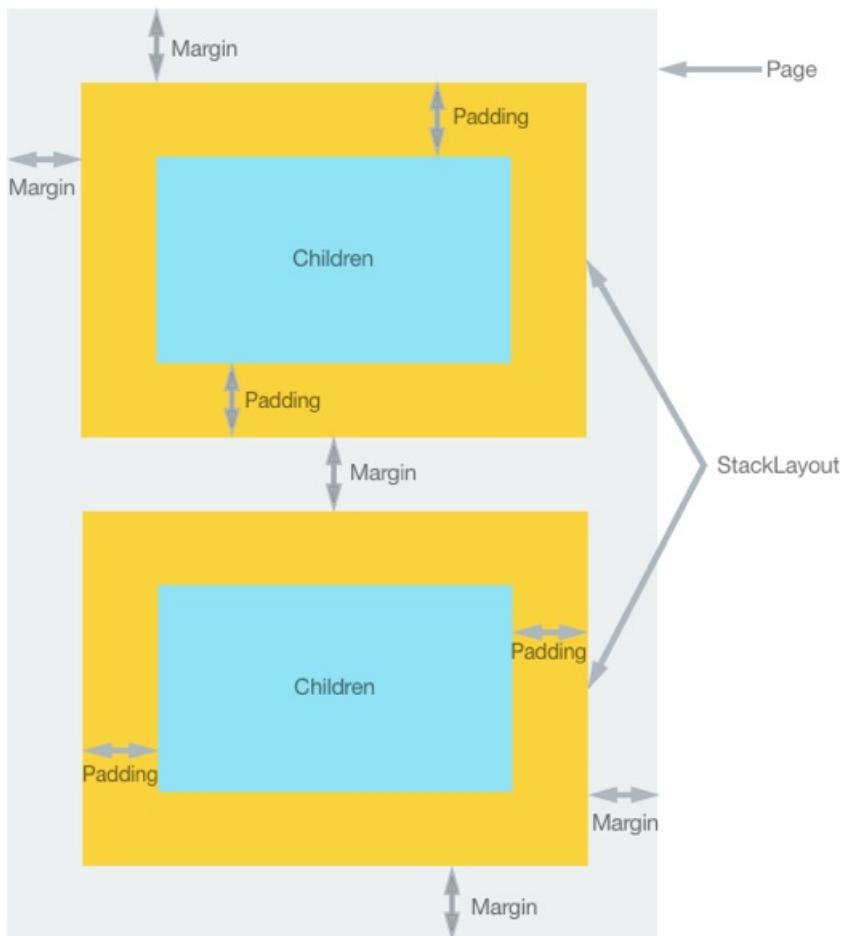
The `Margin` and `Padding` properties control layout behavior when an element is rendered in the user interface. This article demonstrates the difference between the two properties, and how to set them.

## Overview

Margin and padding are related layout concepts:

- The `Margin` property represents the distance between an element and its adjacent elements, and is used to control the element's rendering position, and the rendering position of its neighbors. `Margin` values can be specified on `layout` and `view` classes.
- The `Padding` property represents the distance between an element and its child elements, and is used to separate the control from its own content. `Padding` values can be specified on `layout` classes.

The following diagram illustrates the two concepts:



Note that `Margin` values are additive. Therefore, if two adjacent elements specify a margin of 20 pixels, the distance between the elements will be 40 pixels. In addition, margin and padding are additive when both are applied, in that the distance between an element and any content will be the margin plus padding.

## Specifying a Thickness

The `Margin` and `Padding` properties are both of type `Thickness`. There are three possibilities when creating a

`Thickness` structure:

- Create a `Thickness` structure defined by a single uniform value. The single value is applied to the left, top, right, and bottom sides of the element.
- Create a `Thickness` structure defined by horizontal and vertical values. The horizontal value is symmetrically applied to the left and right sides of the element, with the vertical value being symmetrically applied to the top and bottom sides of the element.
- Create a `Thickness` structure defined by four distinct values that are applied to the left, top, right, and bottom sides of the element.

The following XAML code example shows all three possibilities:

```
<StackLayout Padding="0,20,0,0">
    <Label Text="Xamarin.Forms" Margin="20" />
    <Label Text="Xamarin.iOS" Margin="10, 15" />
    <Label Text="Xamarin.Android" Margin="0, 20, 15, 5" />
</StackLayout>
```

The equivalent C# code is shown in the following code example:

```
var stackLayout = new StackLayout {
    Padding = new Thickness(0,20,0,0),
    Children = {
        new Label { Text = "Xamarin.Forms", Margin = new Thickness (20) },
        new Label { Text = "Xamarin.iOS", Margin = new Thickness (10, 25) },
        new Label { Text = "Xamarin.Android", Margin = new Thickness (0, 20, 15, 5) }
    }
};
```

#### NOTE

`Thickness` values can be negative, which typically clips or overdraws the content.

## Summary

This article demonstrated the difference between the `Margin` and `Padding` properties, and how to set them. The properties control layout behavior when an element is rendered in the user interface.

## Related Links

- [Margin](#)
- [Padding](#)
- [Thickness](#)

# Device Orientation

11/11/2018 • 10 minutes to read • [Edit Online](#)

It is important to consider how your application will be used and how landscape orientation can be incorporated to improve the user experience. Individual layouts can be designed to accommodate multiple orientations and best use the available space. At the application level, rotation can be disabled or enabled.

## Controlling Orientation

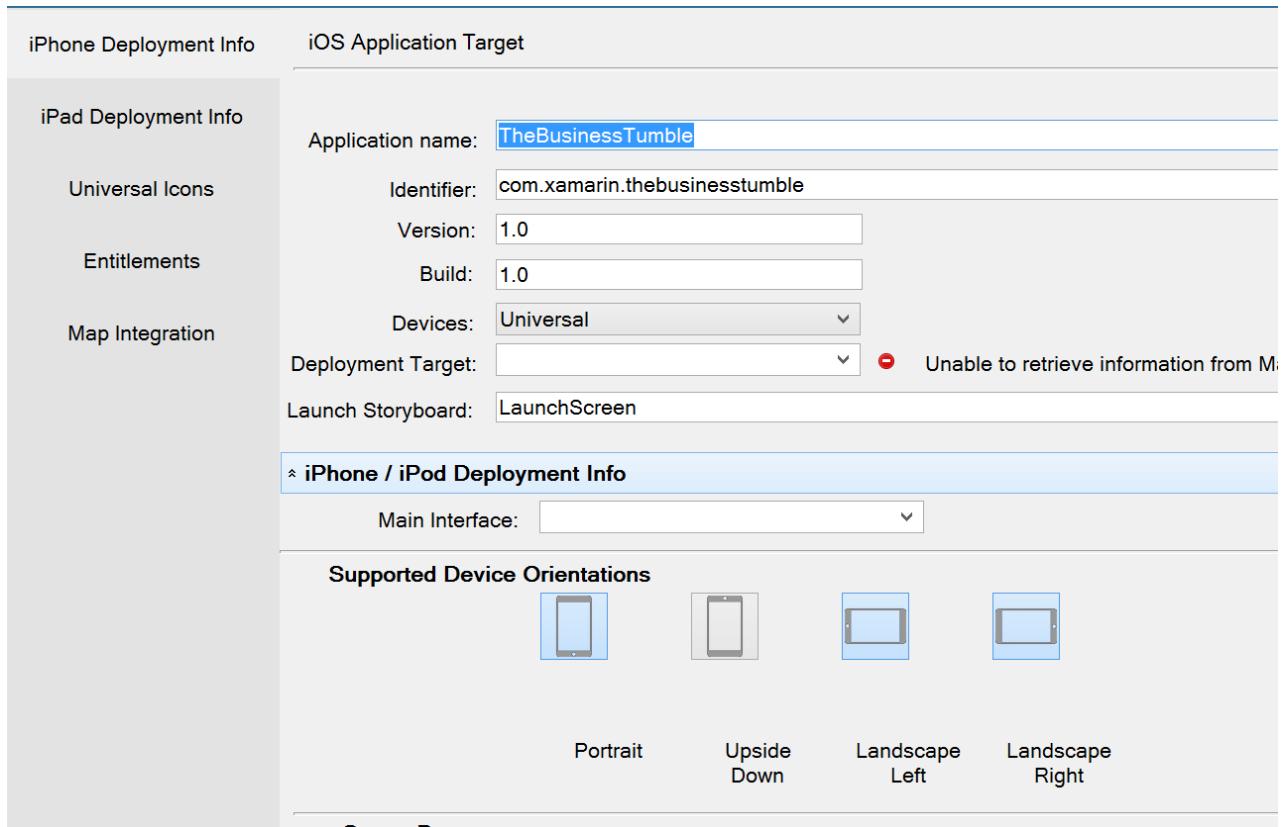
When using Xamarin.Forms, the supported method of controlling device orientation is to use the settings for each individual project.

### iOS

On iOS, device orientation is configured for applications using the **Info.plist** file. This file will include orientation settings for iPhone & iPod, as well as settings for iPad if the app includes it as a target. The following are instructions specific to your IDE. Use the IDE options at the top of this document to select which instructions you'd like to see:

- [Visual Studio](#)
- [Visual Studio for Mac](#)

In Visual Studio, open the iOS project and open **Info.plist**. The file will open into a configuration panel, starting with the iPhone Deployment Info tab:

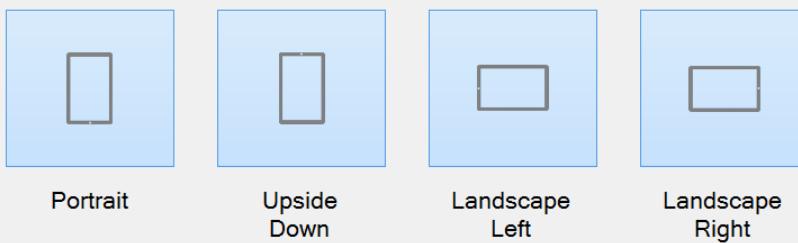


To configure iPad orientation, select the **iPad Deployment Info** tab at the top left of the panel, then select from the available orientations:

## ⌘ iPad Deployment Info

Main Interface:

### Supported Device Orientations



## Android

To control the orientation on Android, open **MainActivity.cs** and set the orientation using the attribute decorating the `MainActivity` class:

```
namespace MyRotatingApp.Droid
{
    [Activity (Label = "MyRotatingApp.Droid", Icon = "@drawable/icon", Theme = "@style/MainTheme",
    MainLauncher = true, ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation,
    ScreenOrientation = ScreenOrientation.Landscape)] //This is what controls orientation
    public class MainActivity : FormsAppCompatActivity
    {
        protected override void OnCreate (Bundle bundle)
    ...
}
```

Xamarin.Android supports several options for specifying orientation:

- **Landscape** – forces the application orientation to be landscape, regardless of sensor data.
- **Portrait** – forces the application orientation to be portrait, regardless of sensor data.
- **User** – causes the application to be presented using the user's preferred orientation.
- **Behind** – causes the application's orientation to be the same as the orientation of the [activity](#) behind it.
- **Sensor** – causes the application's orientation to be determined by the sensor, even if the user has disabled automatic rotation.
- **SensorLandscape** – causes the application to use landscape orientation while using sensor data to change the direction the screen is facing (so that the screen isn't seen as upside down).
- **SensorPortrait** – causes the application to use portrait orientation while using sensor data to change the direction the screen is facing (so that the screen isn't seen as upside down).
- **ReverseLandscape** – causes the application to use landscape orientation, facing the opposite direction from usual, so as to appear "upside down."
- **ReversePortrait** – causes the application to use portrait orientation, facing the opposite direction from usual, so as to appear "upside down."
- **FullSensor** – causes the application to rely on sensor data to select the correct orientation (out of the possible 4).
- **FullUser** – causes the application to use the user's orientation preferences. If automatic rotation is enabled, then all 4 orientations can be used.
- **UserLandscape** – *[Not Supported]* causes the application to use landscape orientation, unless the user has automatic rotation enabled, in which case it will use the sensor to determine orientation. This option will break compilation.
- **UserPortrait** – *[Not Supported]* causes the application to use portrait orientation, unless the user has automatic rotation enabled, in which case it will use the sensor to determine orientation. This option will break compilation.

- **Locked** – [Not Supported] causes the application to use the screen orientation, whatever it is at launch, without responding to changes in the device's physical orientation. This option will break compilation.

Note that the native Android APIs provide a lot of control over how orientation is managed, including options that explicitly contradict the user's expressed preferences.

### Universal Windows platform

On the Universal Windows Platform (UWP), supported orientations are set in the **Package.appxmanifest** file. Opening the manifest will reveal a configuration panel where supported orientations can be selected.

## Reacting to Changes in Orientation

Xamarin.Forms does not offer any native events for notifying your app of orientation changes in shared code. However, the `SizeChanged` event of the `Page` fires when either the width or height of the `Page` changes. When the width of the `Page` is greater than the height, the device is in landscape mode. For more information, see [Display an Image based on Screen Orientation](#).

#### NOTE

There is an existing, free NuGet package for receiving notifications of orientation changes in shared code. See the [GitHub repo](#) for more information.

Alternatively, it's possible to override the `OnSizeAllocated` method on a `Page`, inserting any layout change logic there. The `OnSizeAllocated` method is called whenever a `Page` is allocated a new size, which happens whenever the device is rotated. Note that the base implementation of `OnSizeAllocated` performs important layout functions, so it is important to call the base implementation in the override:

```
protected override void OnSizeAllocated(double width, double height)
{
    base.OnSizeAllocated(width, height); //must be called
}
```

Failure to take that step will result in a non-functioning page.

Note that the `OnSizeAllocated` method may be called many times when a device is rotated. Changing your layout each time is wasteful of resources and can lead to flickering. Consider using an instance variable within your page to track whether the orientation is in landscape or portrait, and only redraw when there is a change:

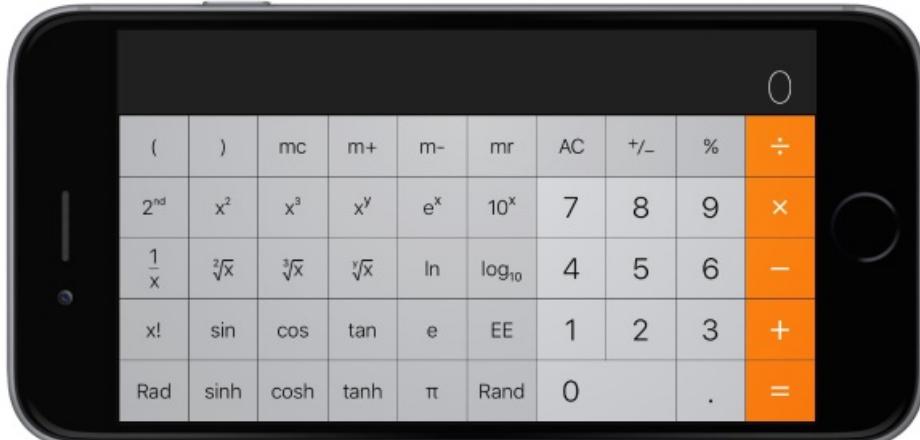
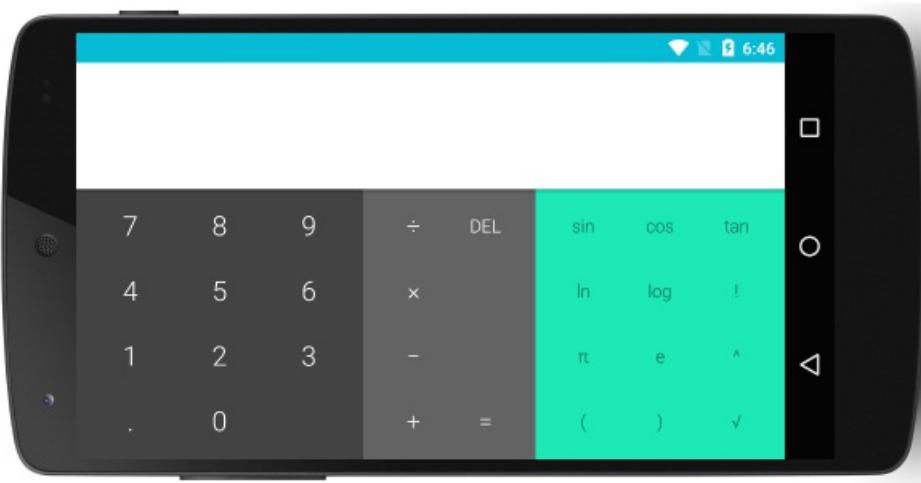
```
private double width = 0;
private double height = 0;

protected override void OnSizeAllocated(double width, double height)
{
    base.OnSizeAllocated(width, height); //must be called
    if (this.width != width || this.height != height)
    {
        this.width = width;
        this.height = height;
        //reconfigure layout
    }
}
```

Once a change in device orientation has been detected, you may want to add or remove additional views to/from your user interface to react to the change in available space. For example, consider the built-in calculator on each platform in portrait:



and landscape:



Notice that the apps take advantage of the available space by adding more functionality in landscape.

## Responsive Layout

It is possible to design interfaces using the built-in layouts so that they transition gracefully when the device is rotated. When designing interfaces that will continue to be appealing when responding to changes in orientation consider the following general rules:

- **Pay attention to ratios** – changes in orientation can cause problems when certain assumptions are made with regards to ratios. For example, a view that would have plenty of space in 1/3 of the vertical space of a screen in portrait may not fit into 1/3 of the vertical space in landscape.
- **Be careful with absolute values** – absolute (pixel) values that make sense in portrait may not make sense in landscape. When absolute values are necessary, use nested layouts to isolate their impact. For example, it would be reasonable to use absolute values in a `TableView` `ItemTemplate` when the item template has a guaranteed uniform height.

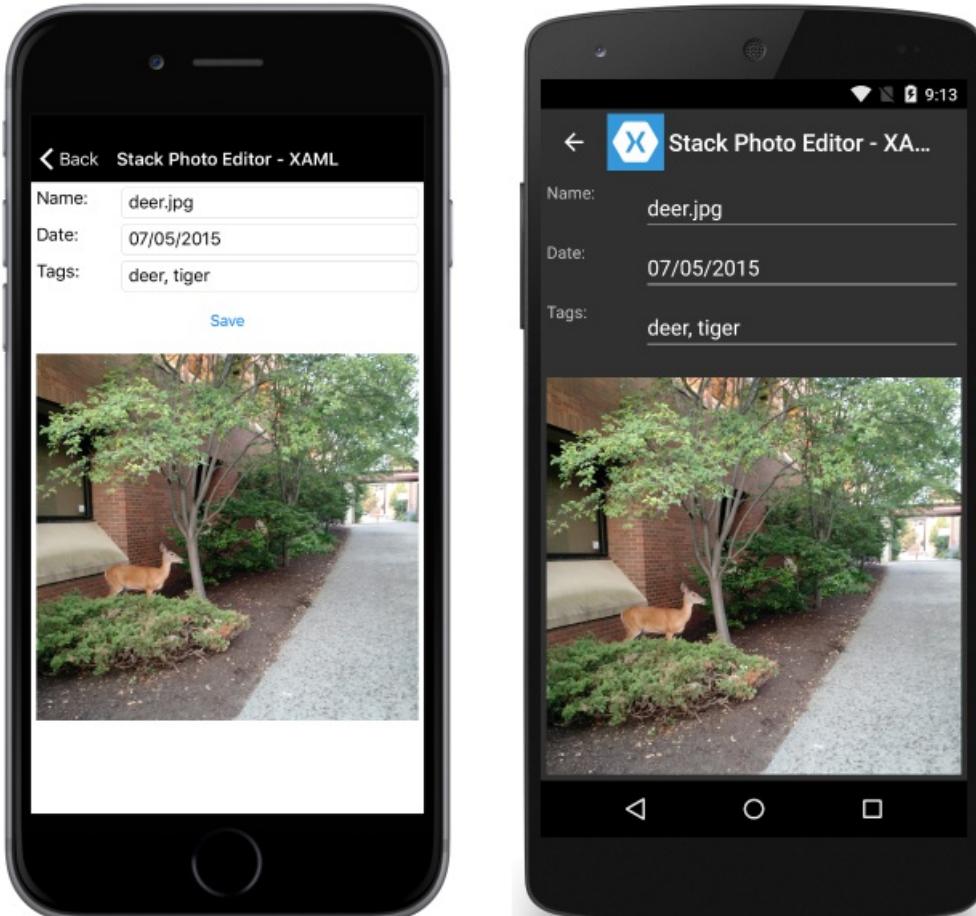
The above rules also apply when implementing interfaces for multiple screen sizes and are generally considered best-practice. The rest of this guide will explain specific examples of responsive layouts using each of the primary layouts in Xamarin.Forms.

### NOTE

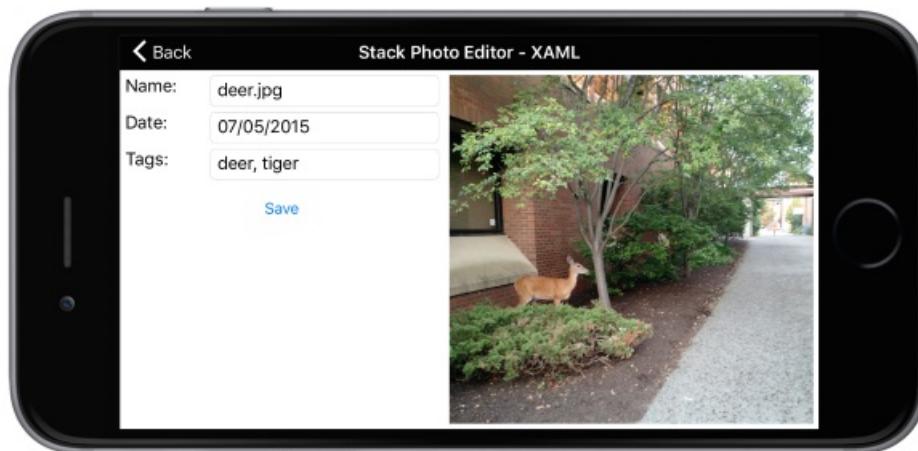
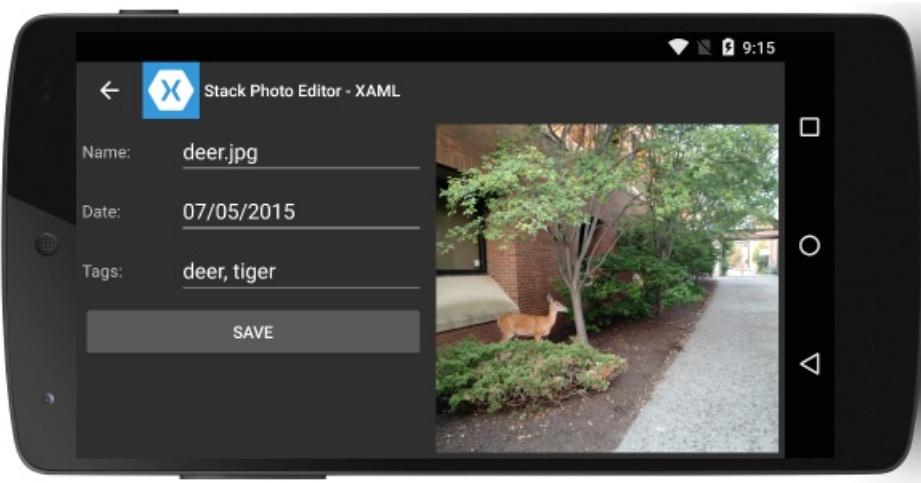
For clarity, the following sections demonstrate how to implement responsive layouts using just one type of `Layout` at a time. In practice, it is often simpler to mix `Layout`s to achieve a desired layout using the simpler or most intuitive `Layout` for each component.

### StackLayout

Consider the following application, displayed in portrait:



and landscape:



That is accomplished with the following XAML:

```

<?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="ResponsiveLayout.StackLayoutPageXaml"
Title="Stack Photo Editor - XAML">
    <ContentPage.Content>
        <StackLayout Spacing="10" Padding="5" Orientation="Vertical"
x:Name="outerStack" > <!-- can change orientation to make responsive -->
            <ScrollView>
                <StackLayout Spacing="5" HorizontalOptions="FillAndExpand"
WidthRequest="1000">
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Name: " WidthRequest="75"
HorizontalOptions="Start" />
                        <Entry Text="deer.jpg"
HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Date: " WidthRequest="75"
HorizontalOptions="Start" />
                        <Entry Text="07/05/2015"
HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Tags:" WidthRequest="75"
HorizontalOptions="Start" />
                        <Entry Text="deer, tiger"
HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Button Text="Save" HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                </StackLayout>
            </ScrollView>
            <Image Source="deer.jpg" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

Some C# is used to change the orientation of `outerStack` based on the orientation of the device:

```

protected override void OnSizeAllocated (double width, double height){
    base.OnSizeAllocated (width, height);
    if (width != this.width || height != this.height) {
        this.width = width;
        this.height = height;
        if (width > height) {
            outerStack.Orientation = StackOrientation.Horizontal;
        } else {
            outerStack.Orientation = StackOrientation.Vertical;
        }
    }
}

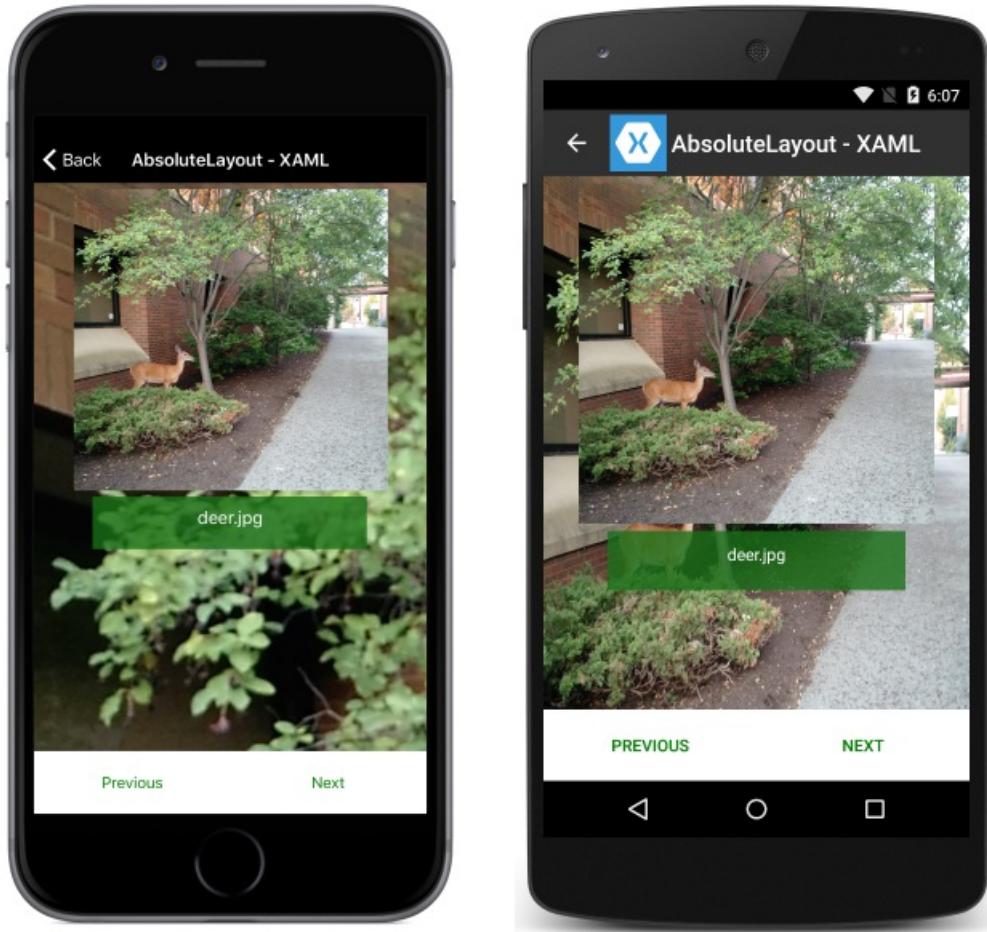
```

Note the following:

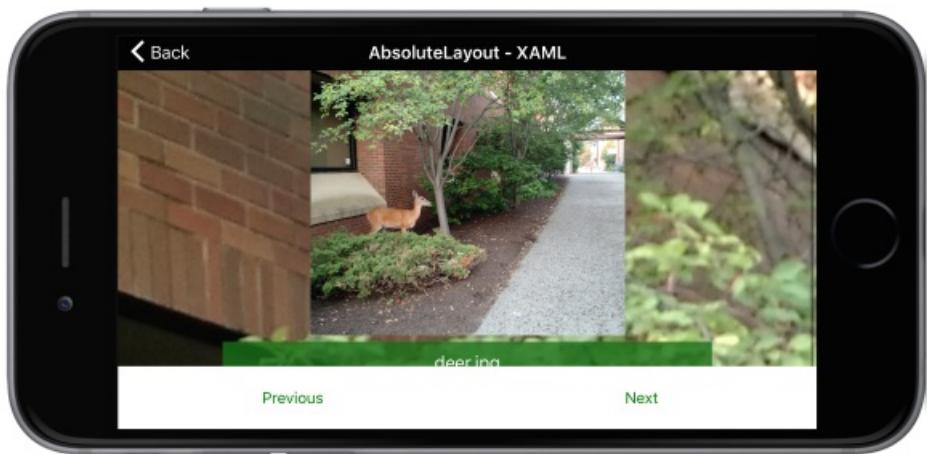
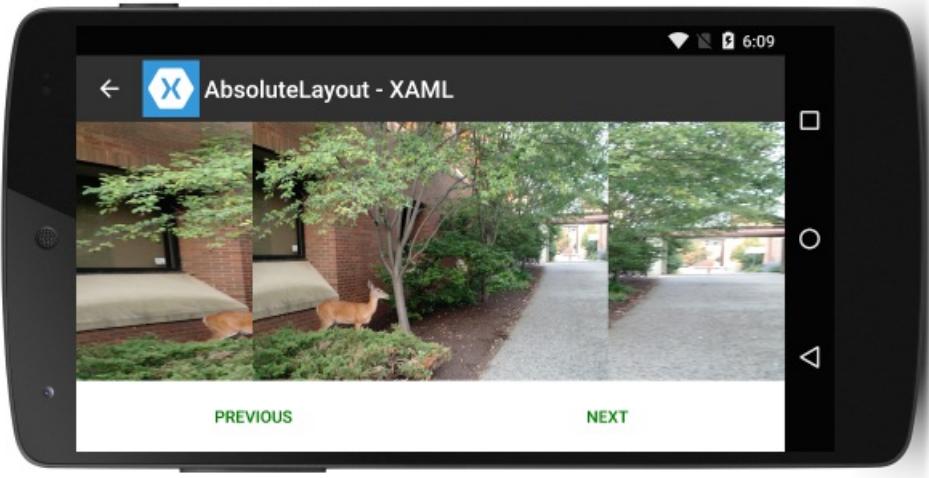
- `outerStack` is adjusted to present the image and controls as a horizontal or vertical stack depending on orientation, to best take advantage of the available space.

## AbsoluteLayout

Consider the following application, displayed in portrait:



and landscape:



That is accomplished with the following XAML:

```

<?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="ResponsiveLayout.AbsoluteLayoutPageXaml"
Title="AbsoluteLayout - XAML" BackgroundImage="deer.jpg">
    <ContentPage.Content>
        <AbsoluteLayout>
            <ScrollView AbsoluteLayout.LayoutBounds="0,0,1,1"
                AbsoluteLayout.LayoutFlags="PositionProportional,SizeProportional">
                <AbsoluteLayout>
                    <Image Source="deer.jpg"
                        AbsoluteLayout.LayoutBounds=".5,0,300,300"
                        AbsoluteLayout.LayoutFlags="PositionProportional" />
                    <BoxView Color="#CC1A7019" AbsoluteLayout.LayoutBounds=".5
                        300,.7,50" AbsoluteLayout.LayoutFlags="XProportional
                        WidthProportional" />
                    <Label Text="deer.jpg" AbsoluteLayout.LayoutBounds = ".5
                        310,1, 50" AbsoluteLayout.LayoutFlags="XProportional
                        WidthProportional" HorizontalTextAlignment="Center" TextColor="White" />
                </AbsoluteLayout>
            </ScrollView>
            <Button Text="Previous" AbsoluteLayout.LayoutBounds="0,1,.5,60"
                AbsoluteLayout.LayoutFlags="PositionProportional
                WidthProportional"
                BackgroundColor="White" TextColor="Green" BorderRadius="0" />
            <Button Text="Next" AbsoluteLayout.LayoutBounds="1,1,.5,60"
                AbsoluteLayout.LayoutFlags="PositionProportional
                WidthProportional" BackgroundColor="White"
                TextColor="Green" BorderRadius="0" />
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>

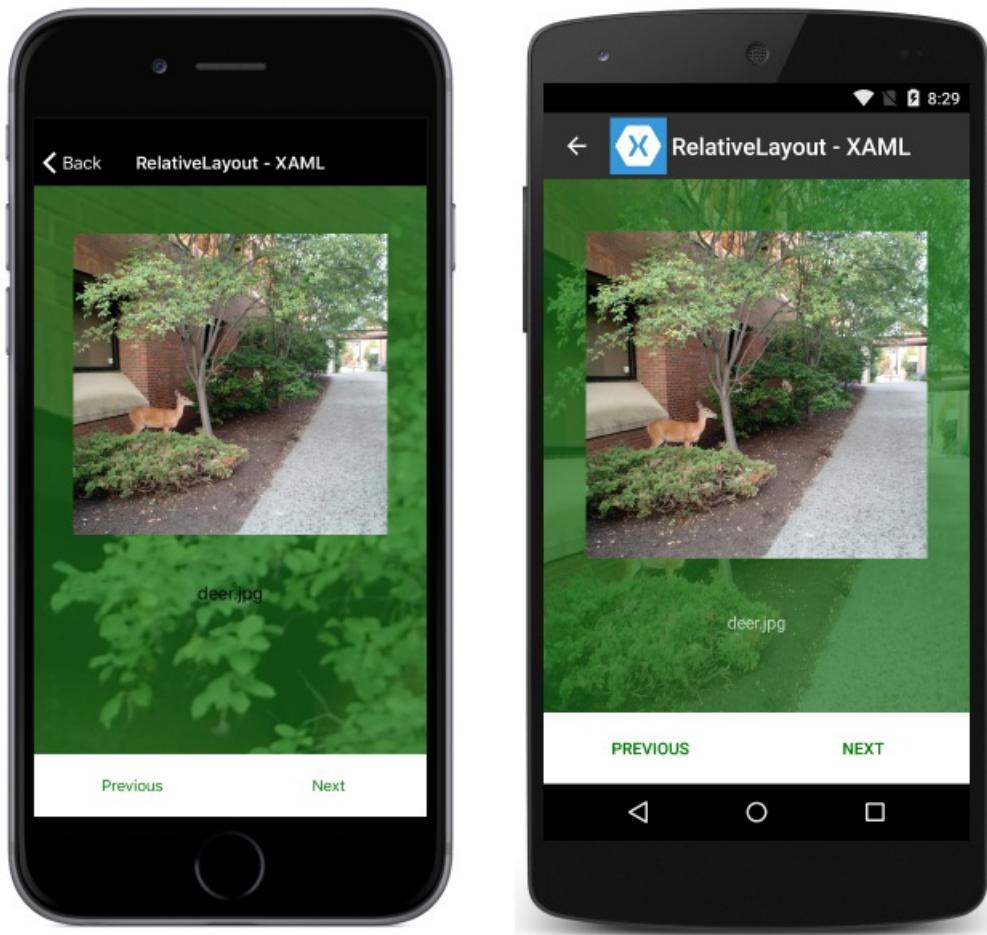
```

Note the following:

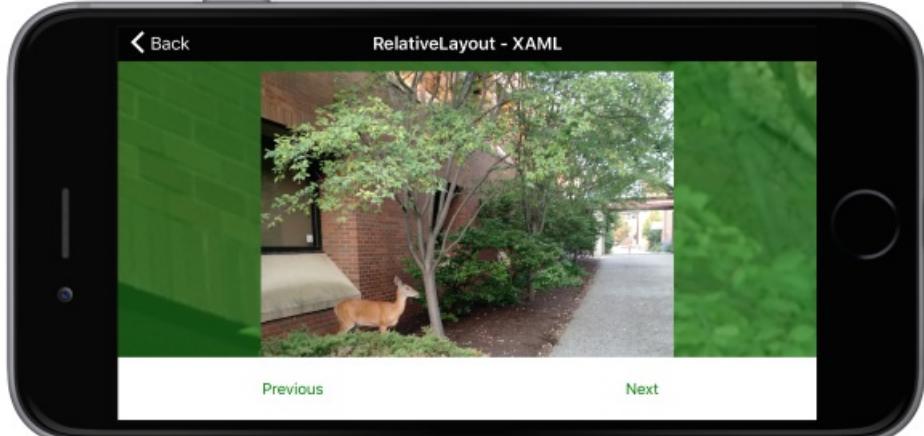
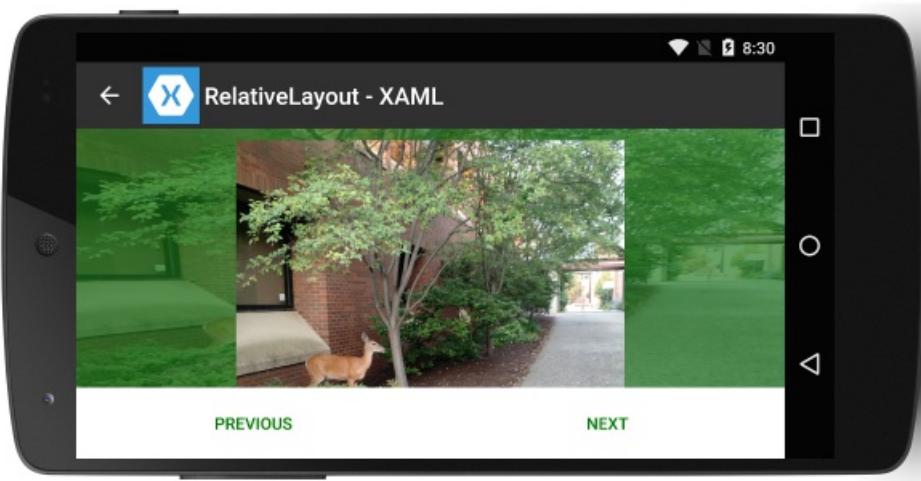
- Because of the way the page has been laid out, there is no need for procedural code to introduce responsiveness.
- The `ScrollView` is being used to allow the label to be visible even when the height of the screen is less than the sum of the fixed heights of the buttons and the image.

## **RelativeLayout**

Consider the following application, displayed in portrait:



and landscape:



That is accomplished with the following XAML:

```
<?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="ResponsiveLayout.RelativeLayoutPageXaml"
  Title="RelativeLayout - XAML"
  BackgroundImage="deer.jpg">
  <ContentPage.Content>
    <RelativeLayout x:Name="outerLayout">
      <BoxView BackgroundColor="#AA1A7019"
        RelativeLayout.WidthConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Width,Factor=1}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Height,Factor=1}"
        RelativeLayout.XConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
        RelativeLayout.YConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Height,Factor=0,Constant=0}" />
      <ScrollView
        RelativeLayout.WidthConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Width,Factor=1}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
        RelativeLayout.XConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
        RelativeLayout.YConstraint="{ConstraintExpression
          Type=RelativeToParent,Property=Height,Factor=0,Constant=0}" />
      <RelativeLayout>
        <Image Source="deer.jpg" x:Name="imageDeer"
          RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.8}"
          RelativeLayout.XConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.1}"
          RelativeLayout.YConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=0,Constant=10}" />
        <Label Text="deer.jpg" HorizontalTextAlignment="Center"
          RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=1}"
          RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=0,Constant=75}"
          RelativeLayout.XConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
          RelativeLayout.YConstraint="{ConstraintExpression
            Type=RelativeToView,ElementName=imageDeer,Property=Height,Factor=1,Constant=20}" />
      />
      </RelativeLayout>
    </ScrollView>

    <Button Text="Previous" BackgroundColor="White" TextColor="Green" BorderRadius="0"
      RelativeLayout.YConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
      RelativeLayout.XConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
      RelativeLayout.HeightConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=0,Constant=60}"
      RelativeLayout.WidthConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=.5}"
      />
    <Button Text="Next" BackgroundColor="White" TextColor="Green" BorderRadius="0"
      RelativeLayout.XConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=.5}"
      RelativeLayout.YConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
      RelativeLayout.HeightConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=0,Constant=60}"
      RelativeLayout.WidthConstraint="{ConstraintExpression
        Type=RelativeToParent,Property=Width,Factor=1}" />
  
```

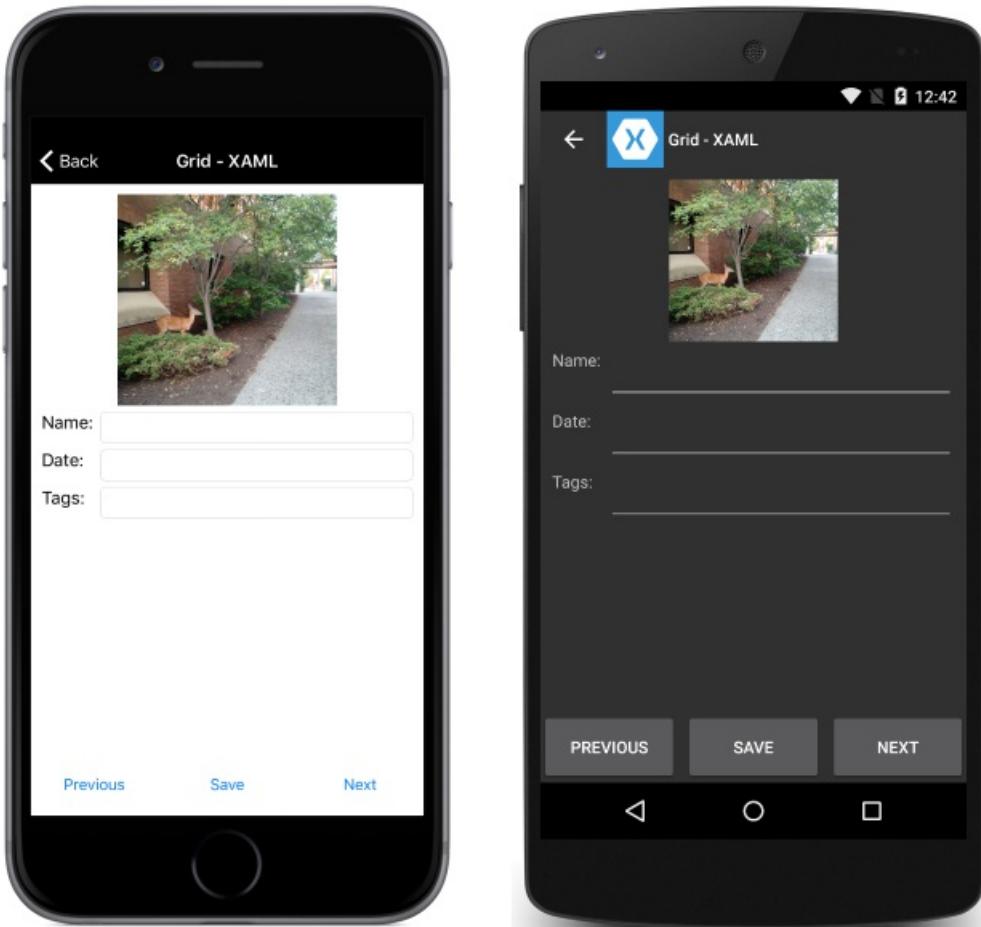
```
Type=RelativeToParent,Property=Width,Factor=.5}"  
/>>  
</RelativeLayout>  
</ContentPage.Content>  
</ContentPage>
```

Note the following:

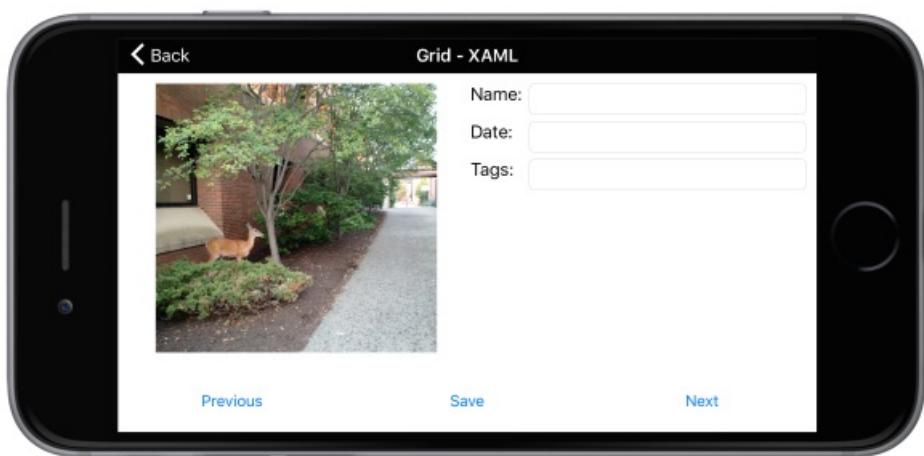
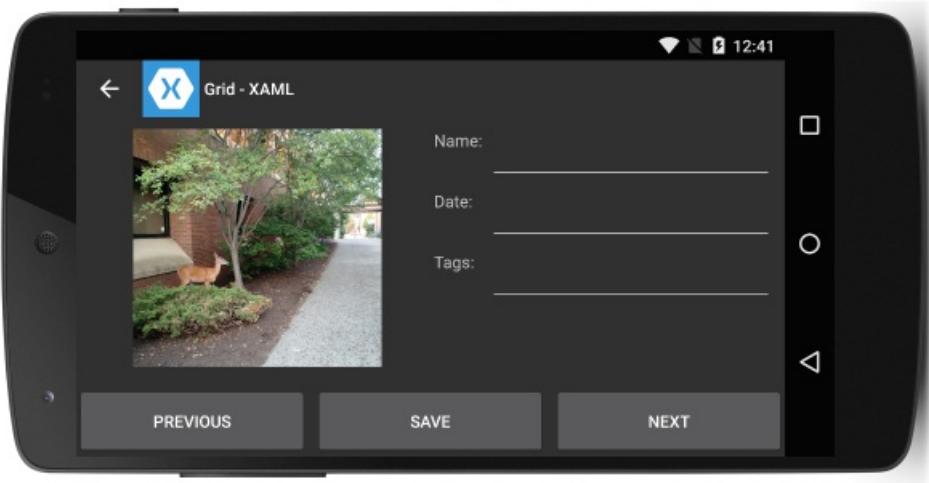
- Because of the way the page has been laid out, there is no need for procedural code to introduce responsiveness.
- The `ScrollView` is being used to allow the label to be visible even when the height of the screen is less than the sum of the fixed heights of the buttons and the image.

## Grid

Consider the following application, displayed in portrait:



and landscape:



That is accomplished with the following XAML:

```

<?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="ResponsiveLayout.GridPageXaml"
Title="Grid - XAML">
    <ContentPage.Content>
        <Grid x:Name="outerGrid">
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="60" />
            </Grid.RowDefinitions>
            <Grid x:Name="innerGrid" Grid.Row="0" Padding="10">
                <Grid.RowDefinitions>
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Image Source="deer.jpg" Grid.Row="0" Grid.Column="0" HeightRequest="300" WidthRequest="300"
/>
                <Grid x:Name="controlsGrid" Grid.Row="0" Grid.Column="1" >
                    <Grid.RowDefinitions>
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
                    <Label Text="Name:" Grid.Row="0" Grid.Column="0" />
                    <Label Text="Date:" Grid.Row="1" Grid.Column="0" />
                    <Label Text="Tags:" Grid.Row="2" Grid.Column="0" />
                    <Entry Grid.Row="0" Grid.Column="1" />
                    <Entry Grid.Row="1" Grid.Column="1" />
                    <Entry Grid.Row="2" Grid.Column="1" />
                </Grid>
            </Grid>
            <Grid x:Name="buttonsGrid" Grid.Row="1">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Button Text="Previous" Grid.Column="0" />
                <Button Text="Save" Grid.Column="1" />
                <Button Text="Next" Grid.Column="2" />
            </Grid>
        </Grid>
    </ContentPage.Content>
</ContentPage>

```

Along with the following procedural code to handle rotation changes:

```

private double width;
private double height;

protected override void OnSizeAllocated (double width, double height){
    base.OnSizeAllocated (width, height);
    if (width != this.width || height != this.height) {
        this.width = width;
        this.height = height;
        if (width > height) {
            innerGrid.RowDefinitions.Clear();
            innerGrid.ColumnDefinitions.Clear ();
            innerGrid.RowDefinitions.Add (new RowDefinition{ Height = new GridLength (1, GridUnitType.Star)
        });
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.Children.Remove (controlsGrid);
            innerGrid.Children.Add (controlsGrid, 1, 0);
        } else {
            innerGrid.ColumnDefinitions.Clear ();
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition{ Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Auto)
        });
            innerGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star)
        });
            innerGrid.Children.Remove (controlsGrid);
            innerGrid.Children.Add (controlsGrid, 0, 1);
        }
    }
}

```

Note the following:

- Because of the way the page has been laid out, there is a method to change the grid placement of the controls.

## Related Links

- [Layout \(sample\)](#)
- [BusinessTumble Example \(sample\)](#)
- [Responsive Layout \(sample\)](#)
- [Display an Image based on Screen Orientation](#)

# Layout for Tablet and Desktop apps

7/12/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms supports all device types available on the supported platforms, so in addition to phones, apps can also run on:

- iPads,
- Android tablets,
- Windows tablets and desktop computers (running Windows 10).

This page briefly discusses:

- the supported [device types](#), and
- how to [optimize](#) layouts for tablets versus phones.

## Device Types

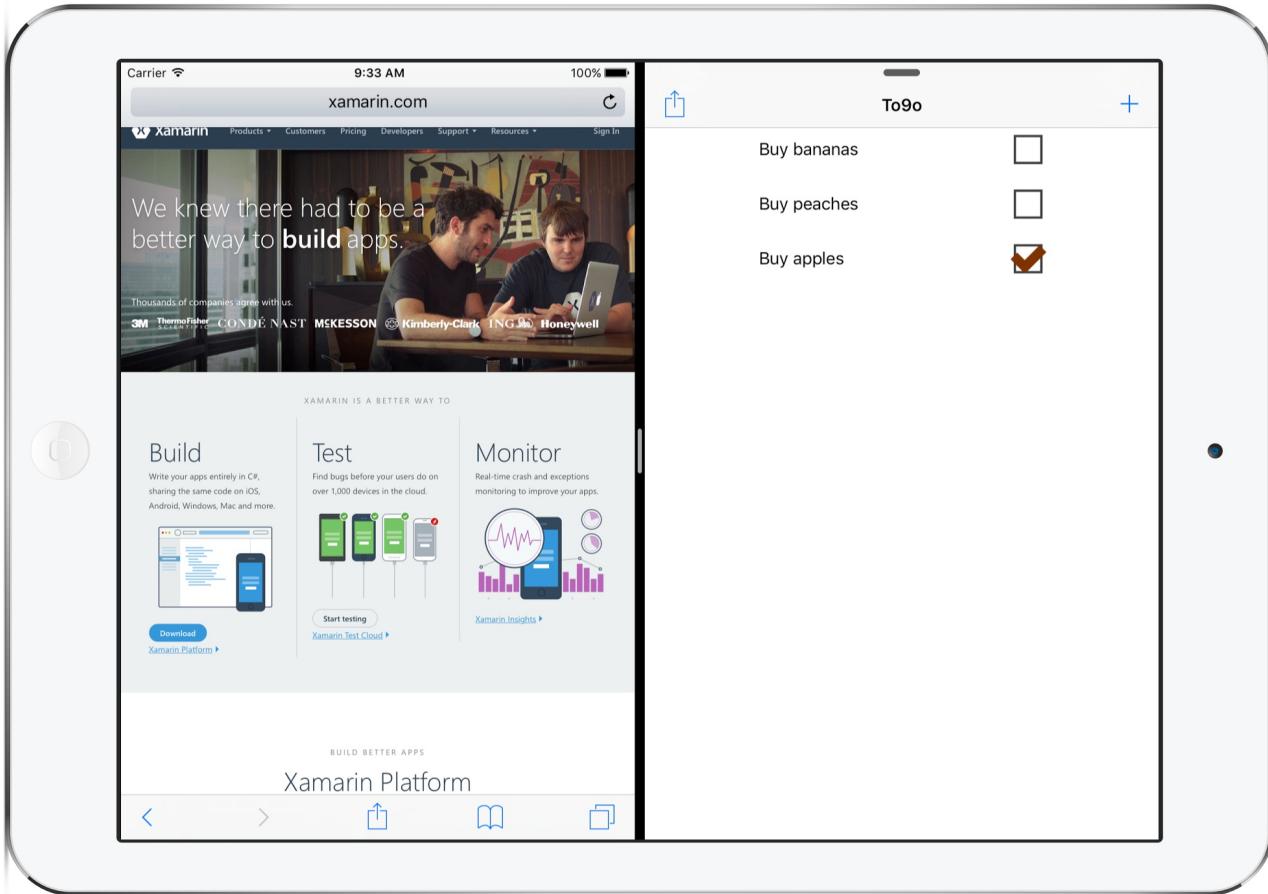
Larger screen devices are available for all of the platforms supported by Xamarin.Forms.

### iPads (iOS)

The Xamarin.Forms template automatically includes iPad support by configuring the **Info.plist > Devices** setting to **Universal** (which means both iPhone and iPad are supported).

To provide a pleasant startup experience, and ensure the full screen resolution is used on all devices, you should make sure an [iPad-specific launch screen](#) (using a storyboard) is provided. This ensures the app is rendered correctly on iPad mini, iPad, and iPad Pro devices.

Prior to iOS 9 all apps took up the full screen on the device, but some iPads can now perform [split screen multitasking](#). This means your app could take up just a slim column on the side of the screen, 50% of the width of the screen, or the entire screen.



Split-screen functionality means you should design your app to work well with as little as 320 pixels wide, or as much as 1366 pixels wide.

## Android Tablets

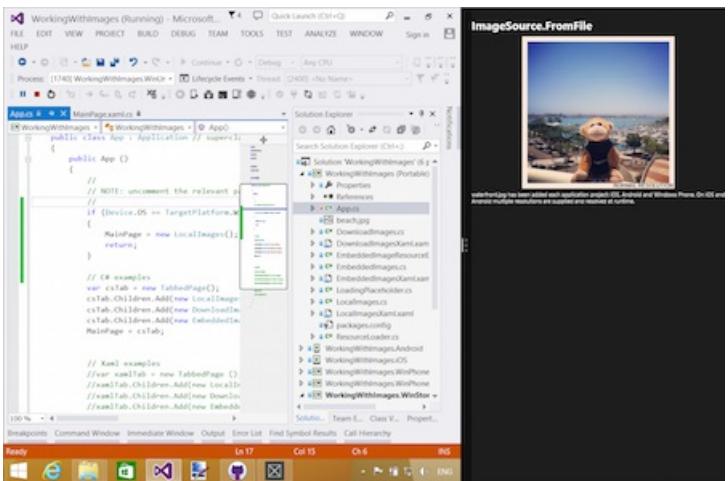
The Android ecosystem has a myriad of supported screen sizes, from small phones up to large tablets. Xamarin.Forms can support all screen sizes, but as with the other platforms you might want to adjust your user interface for larger devices.

When supporting many different screen resolutions, you can provide your native image resources in different sizes to optimize the user experience. Review the [Android resources](#) documentation (and in particular [creating resources for varying screen sizes](#)) for more information on how to structure the folders and filenames in your Android app project to include optimized image resources in your app.

## Windows Tablets and Desktops

To support tablets and desktop computers running Windows, you'll need to use [Windows UWP support](#), which builds universal apps that run on Windows 10.

Apps running on Windows tablets and desktops can be resized to arbitrary dimensions in addition to running full-screen.



## Optimizing for Tablet and Desktop

You can adjust your Xamarin.Forms user interface depending on whether a phone or tablet/desktop device is being used. This means you can optimize the user-experience for large-screen devices such as tablets and desktop computers.

### Device.Idiom

You can use the `Device` class to change the behavior of your app or user interface. Using the `Device.Idiom` enumeration you can

```
if (Device.Idiom == TargetIdiom.Phone)
{
    HeroImage.Source = ImageSource.FromFile("hero.jpg");
} else {
    HeroImage.Source = ImageSource.FromFile("herotablet.jpg");
}
```

This approach can be expanded to make significant changes to individual page layouts, or even to render entirely different pages on larger screens.

### Leveraging MasterDetailPage

The `MasterDetailPage` is ideal for larger screens, especially on the iPad where it uses the `UISplitViewController` to provide a native iOS experience.

Review [this Xamarin blog post](#) to see how you can adapt your user interface so that phones use one layout and larger screens can use another (with the `MasterDetailPage`).

## Related Links

- [Xamarin Blog](#)
- [MyShoppe sample](#)

# Creating a Custom Layout

7/12/2018 • 14 minutes to read • [Edit Online](#)

Xamarin.Forms defines four layout classes – `StackLayout`, `AbsoluteLayout`, `RelativeLayout`, and `Grid`, and each arranges its children in a different way. However, sometimes it's necessary to organize page content using a layout not provided by Xamarin.Forms. This article explains how to write a custom layout class, and demonstrates an orientation-sensitive `WrapLayout` class that arranges its children horizontally across the page, and then wraps the display of subsequent children to additional rows.

## Overview

In Xamarin.Forms, all layout classes derive from the `Layout<T>` class and constrain the generic type to `View` and its derived types. In turn, the `Layout<T>` class derives from the `Layout` class, which provides the mechanism for positioning and sizing child elements.

Every visual element is responsible for determining its own preferred size, which is known as the *requested* size. `Page`, `Layout`, and `Layout<View>` derived types are responsible for determining the location and size of their child, or children, relative to themselves. Therefore, layout involves a parent-child relationship, where the parent determines what the size of its children should be, but will attempt to accommodate the requested size of the child.

A thorough understanding of the Xamarin.Forms layout and invalidation cycles is required to create a custom layout. These cycles will now be discussed.

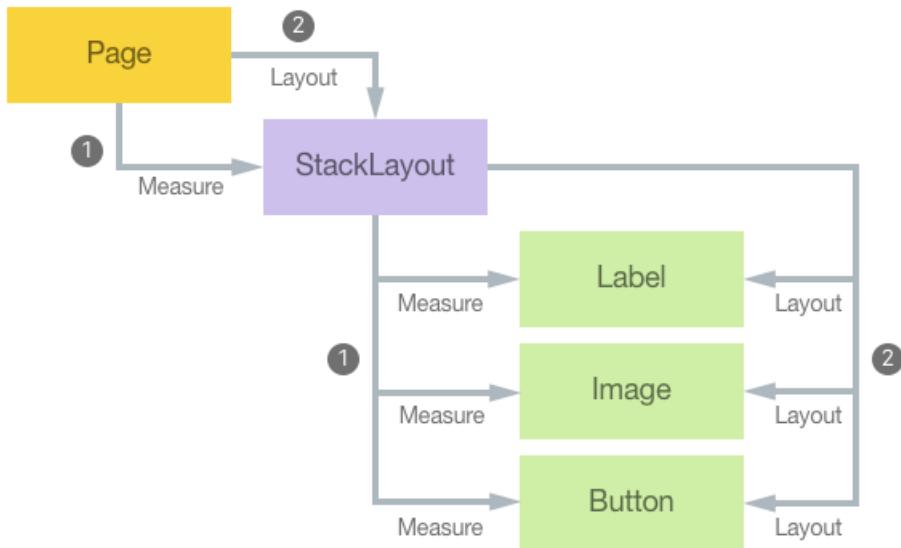
## Layout

Layout begins at the top of the visual tree with a page, and it proceeds through all branches of the visual tree to encompass every visual element on a page. Elements that are parents to other elements are responsible for sizing and positioning their children relative to themselves.

The `VisualElement` class defines a `Measure` method that measures an element for layout operations, and a `Layout` method that specifies the rectangular area the element will be rendered within. When an application starts and the first page is displayed, a *layout cycle* consisting first of `Measure` calls, and then `Layout` calls, starts on the `Page` object:

1. During the layout cycle, every parent element is responsible for calling the `Measure` method on its children.
2. After the children have been measured, every parent element is responsible for calling the `Layout` method on its children.

This cycle ensures that every visual element on the page receives calls to the `Measure` and `Layout` methods. The process is shown in the following diagram:



#### NOTE

Note that layout cycles can also occur on a subset of the visual tree if something changes to affect the layout. This includes items being added or removed from a collection such as in a `StackLayout`, a change in the `IsVisible` property of an element, or a change in the size of an element.

Every `Xamarin.Forms` class that has a `Content` or a `Children` property has an overridable `LayoutChildren` method. Custom layout classes that derive from `Layout<View>` must override this method and ensure that the `Measure` and `Layout` methods are called on all the element's children, to provide the desired custom layout.

In addition, every class that derives from `Layout` or `Layout<View>` must override the `OnMeasure` method, which is where a layout class determines the size that it needs to be by making calls to the `Measure` methods of its children.

#### NOTE

Elements determine their size based on *constraints*, which indicate how much space is available for an element within the element's parent. Constraints passed to the `Measure` and `OnMeasure` methods can range from 0 to `Double.PositiveInfinity`. An element is *constrained*, or *fully constrained*, when it receives a call to its `Measure` method with non-infinite arguments - the element is constrained to a particular size. An element is *unconstrained*, or *partially constrained*, when it receives a call to its `Measure` method with at least one argument equal to `Double.PositiveInfinity` – the infinite constraint can be thought of as indicating autosizing.

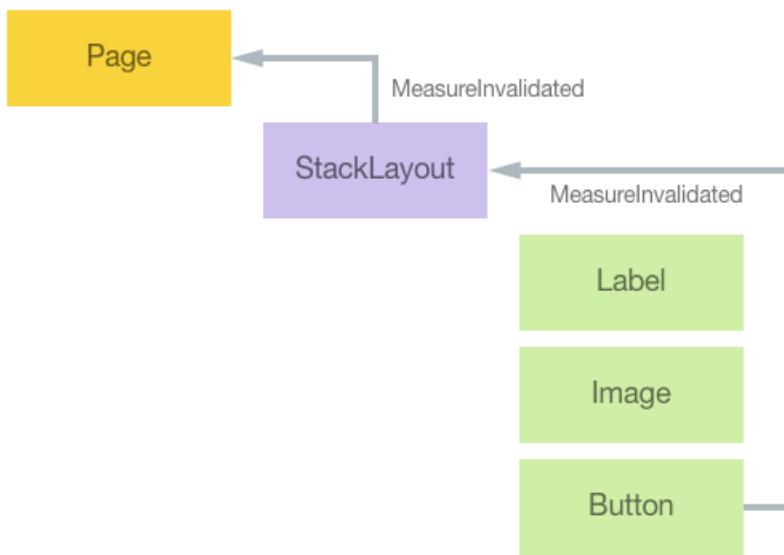
## Invalidation

Invalidation is the process by which a change in an element on a page triggers a new layout cycle. Elements are considered invalid when they no longer have the correct size or position. For example, if the `FontSize` property of a `Button` changes, the `Button` is said to be invalid because it will no longer have the correct size. Resizing the `Button` may then have a ripple effect of changes in layout through the rest of a page.

Elements invalidate themselves by invoking the `InvalidateMeasure` method, generally when a property of the element changes that might result in a new size of the element. This method fires the `MeasureInvalidated` event, which the element's parent handles to trigger a new layout cycle.

The `Layout` class sets a handler for the `MeasureInvalidated` event on every child added to its `Content` property or `Children` collection, and detaches the handler when the child is removed. Therefore, every element in the visual tree that has children is alerted whenever one of its children changes size. The following diagram illustrates

how a change in the size of an element in the visual tree can cause changes that ripple up the tree:



However, the `Layout` class attempts to restrict the impact of a change in a child's size on the layout of a page. If the layout is size constrained, then a child size change does not affect anything higher than the parent layout in the visual tree. However, usually a change in the size of a layout affects how the layout arranges its children. Therefore, any change in a layout's size will start a layout cycle for the layout, and the layout will receive calls to its `OnMeasure` and `LayoutChildren` methods.

The `Layout` class also defines an `InvalidateLayout` method that has a similar purpose to the `InvalidateMeasure` method. The `InvalidateLayout` method should be invoked whenever a change is made that affects how the layout positions and sizes its children. For example, the `Layout` class invokes the `InvalidateLayout` method whenever a child is added to or removed from a layout.

The `InvalidateLayout` can be overridden to implement a cache to minimize repetitive invocations of the `Measure` methods of the layout's children. Overriding the `InvalidateLayout` method will provide a notification of when children are added to or removed from the layout. Similarly, the `OnChildMeasureInvalidated` method can be overridden to provide a notification when one of the layout's children changes size. For both method overrides, a custom layout should respond by clearing the cache. For more information, see [Calculating and Caching Data](#).

## Creating a Custom Layout

The process for creating a custom layout is as follows:

1. Create a class that derives from the `Layout<View>` class. For more information, see [Creating a WrapLayout](#).
2. [optional] Add properties, backed by bindable properties, for any parameters that should be set on the layout class. For more information, see [Adding Properties Backed by Bindable Properties](#).
3. Override the `OnMeasure` method to invoke the `Measure` method on all the layout's children, and return a requested size for the layout. For more information, see [Overriding the OnMeasure Method](#).
4. Override the `LayoutChildren` method to invoke the `Layout` method on all the layout's children. Failure to invoke the `Layout` method on each child in a layout will result in the child never receiving a correct size or position, and hence the child will not become visible on the page. For more information, see [Overriding the LayoutChildren Method](#).

### NOTE

When enumerating children in the `OnMeasure` and `LayoutChildren` overrides, skip any child whose `IsVisible` property is set to `false`. This will ensure that the custom layout won't leave space for invisible children.

- [optional] Override the `InvalidateLayout` method to be notified when children are added to or removed from the layout. For more information, see [Overriding the InvalidateLayout Method](#).
- [optional] Override the `OnChildMeasureInvalidated` method to be notified when one of the layout's children changes size. For more information, see [Overriding the OnChildMeasureInvalidated Method](#).

#### NOTE

Note that the `OnMeasure` override won't be invoked if the size of the layout is governed by its parent, rather than its children. However, the override will be invoked if one or both of the constraints are infinite, or if the layout class has non-default `HorizontalOptions` or `VerticalOptions` property values. For this reason, the `LayoutChildren` override can't rely on child sizes obtained during the `OnMeasure` method call. Instead, `LayoutChildren` must invoke the `Measure` method on the layout's children, before invoking the `Layout` method. Alternatively, the size of the children obtained in the `OnMeasure` override can be cached to avoid later `Measure` invocations in the `LayoutChildren` override, but the layout class will need to know when the sizes need to be obtained again. For more information, see [Calculating and Caching Layout Data](#).

The layout class can then be consumed by adding it to a `Page`, and by adding children to the layout. For more information, see [Consuming the WrapLayout](#).

#### Creating a WrapLayout

The sample application demonstrates an orientation-sensitive `WrapLayout` class that arranges its children horizontally across the page, and then wraps the display of subsequent children to additional rows.

The `WrapLayout` class allocates the same amount of space for each child, known as the *cell size*, based on the maximum size of the children. Children smaller than the cell size can be positioned within the cell based on their `HorizontalOptions` and `VerticalOptions` property values.

The `WrapLayout` class definition is shown in the following code example:

```
public class WrapLayout : Layout<View>
{
    Dictionary<Size, LayoutData> layoutDataCache = new Dictionary<Size, LayoutData>();
    ...
}
```

#### Calculating and Caching Layout Data

The `LayoutData` structure stores data about a collection of children in a number of properties:

- `VisibleChildCount` – the number of children that are visible in the layout.
- `CellSize` – the maximum size of all the children, adjusted to the size of the layout.
- `Rows` – the number of rows.
- `Columns` – the number of columns.

The `layoutDataCache` field is used to store multiple `LayoutData` values. When the application starts, two `LayoutData` objects will be cached into the `layoutDataCache` dictionary for the current orientation – one for the constraint arguments to the `OnMeasure` override, and one for the `width` and `height` arguments to the `LayoutChildren` override. When rotating the device into landscape orientation, the `OnMeasure` override and the `LayoutChildren` override will again be invoked, which will result in another two `LayoutData` objects being cached into the dictionary. However, when returning the device to portrait orientation, no further calculations are required because the `layoutDataCache` already has the required data.

The following code example shows the `GetLayoutData` method, which calculates the properties of the `LayoutData` structured based on a particular size:

```

LayoutData GetLayoutData(double width, double height)
{
    Size size = new Size(width, height);

    // Check if cached information is available.
    if (layoutDataCache.ContainsKey(size))
    {
        return layoutDataCache[size];
    }

    int visibleChildCount = 0;
    Size maxChildSize = new Size();
    int rows = 0;
    int columns = 0;
    LayoutData layoutData = new LayoutData();

    // Enumerate through all the children.
    foreach (View child in Children)
    {
        // Skip invisible children.
        if (!child.IsVisible)
            continue;

        // Count the visible children.
        visibleChildCount++;

        // Get the child's requested size.
        SizeRequest childSizeRequest = child.Measure(Double.PositiveInfinity, Double.PositiveInfinity);

        // Accumulate the maximum child size.
        maxChildSize.Width = Math.Max(maxChildSize.Width, childSizeRequest.Request.Width);
        maxChildSize.Height = Math.Max(maxChildSize.Height, childSizeRequest.Request.Height);
    }

    if (visibleChildCount != 0)
    {
        // Calculate the number of rows and columns.
        if (Double.IsPositiveInfinity(width))
        {
            columns = visibleChildCount;
            rows = 1;
        }
        else
        {
            columns = (int)((width + ColumnSpacing) / (maxChildSize.Width + ColumnSpacing));
            columns = Math.Max(1, columns);
            rows = (visibleChildCount + columns - 1) / columns;
        }

        // Now maximize the cell size based on the layout size.
        Size cellSize = new Size();

        if (Double.IsPositiveInfinity(width))
            cellSize.Width = maxChildSize.Width;
        else
            cellSize.Width = (width - ColumnSpacing * (columns - 1)) / columns;

        if (Double.IsPositiveInfinity(height))
            cellSize.Height = maxChildSize.Height;
        else
            cellSize.Height = (height - RowSpacing * (rows - 1)) / rows;

        layoutData = new LayoutData(visibleChildCount, cellSize, rows, columns);
    }

    layoutDataCache.Add(size, layoutData);
    return layoutData;
}

```

The `GetLayoutData` method performs the following operations:

- It determines whether a calculated `LayoutData` value is already in the cache and returns it if it's available.
- Otherwise, it enumerates through all the children, invoking the `Measure` method on each child with an infinite width and height, and determines the maximum child size.
- Provided that there's at least one visible child, it calculates the number of rows and columns required, and then calculates a cell size for the children based on the dimensions of the `WrapLayout`. Note that the cell size is usually slightly wider than the maximum child size, but that it could also be smaller if the `WrapLayout` isn't wide enough for the widest child or tall enough for the tallest child.
- It stores the new `LayoutData` value in the cache.

#### Adding Properties Backed by Bindable Properties

The `WrapLayout` class defines `ColumnSpacing` and `RowSpacing` properties, whose values are used to separate the rows and columns in the layout, and which are backed by bindable properties. The bindable properties are shown in the following code example:

```
public static readonly BindableProperty ColumnSpacingProperty = BindableProperty.Create(
    "ColumnSpacing",
    typeof(double),
    typeof(WrapLayout),
    5.0,
    propertyChanged: (bindable, oldvalue, newvalue) =>
{
    ((WrapLayout)bindable).InvalidateLayout();
});

public static readonly BindableProperty RowSpacingProperty = BindableProperty.Create(
    "RowSpacing",
    typeof(double),
    typeof(WrapLayout),
    5.0,
    propertyChanged: (bindable, oldvalue, newvalue) =>
{
    ((WrapLayout)bindable).InvalidateLayout();
});
```

The property-changed handler of each bindable property invokes the `InvalidateLayout` method override to trigger a new layout pass on the `WrapLayout`. For more information, see [Overriding the InvalidateLayout Method](#) and [Overriding the OnChildMeasureInvalidated Method](#).

#### Overriding the OnMeasure Method

The `OnMeasure` override is shown in the following code example:

```
protected override SizeRequest OnMeasure(double widthConstraint, double heightConstraint)
{
    LayoutData layoutData = GetLayoutData(widthConstraint, heightConstraint);
    if (layoutData.VisibleChildCount == 0)
    {
        return new SizeRequest();
    }

    Size totalSize = new Size(layoutData.CellSize.Width * layoutData.Columns + ColumnSpacing *
(layoutData.Columns - 1),
    layoutData.CellSize.Height * layoutData.Rows + RowSpacing * (layoutData.Rows - 1));
    return new SizeRequest(totalSize);
}
```

The override invokes the `GetLayoutData` method and constructs a `SizeRequest` object from the returned data,

while also taking into account the `RowSpacing` and `ColumnSpacing` property values. For more information about the `GetLayoutData` method, see [Calculating and Caching Data](#).

#### IMPORTANT

The `Measure` and `OnMeasure` methods should never request an infinite dimension by returning a `SizeRequest` value with a property set to `Double.PositiveInfinity`. However, at least one of the constraint arguments to `OnMeasure` can be `Double.PositiveInfinity`.

#### Overriding the `LayoutChildren` Method

The `LayoutChildren` override is shown in the following code example:

```
protected override void LayoutChildren(double x, double y, double width, double height)
{
    LayoutData layoutData = GetLayoutData(width, height);

    if (layoutData.VisibleChildCount == 0)
    {
        return;
    }

    double xChild = x;
    double yChild = y;
    int row = 0;
    int column = 0;

    foreach (View child in Children)
    {
        if (!child.isVisible)
        {
            continue;
        }

        LayoutChildIntoBoundingRegion(child, new Rectangle(new Point(xChild, yChild), layoutData.CellSize));
        if (++column == layoutData.Columns)
        {
            column = 0;
            row++;
            xChild = x;
            yChild += RowSpacing + layoutData.CellSize.Height;
        }
        else
        {
            xChild += ColumnSpacing + layoutData.CellSize.Width;
        }
    }
}
```

The override begins with a call to the `GetLayoutData` method, and then enumerates all of the children to size and position them within each child's cell. This is achieved by invoking the `LayoutChildIntoBoundingRegion` method, which is used to position a child within a rectangle based on its `HorizontalOptions` and `VerticalOptions` property values. This is equivalent to making a call to the child's `Layout` method.

#### NOTE

Note that the rectangle passed to the `LayoutChildIntoBoundingRegion` method includes the whole area in which the child can reside.

For more information about the `GetLayoutData` method, see [Calculating and Caching Data](#).

## Overriding the `InvalidateLayout` Method

The `InvalidateLayout` override is invoked when children are added to or removed from the layout, or when one of the `WrapLayout` properties changes value, as shown in the following code example:

```
protected override void InvalidateLayout()
{
    base.InvalidateLayout();
    layoutInfoCache.Clear();
}
```

The override invalidates the layout and discards all the cached layout information.

### NOTE

To stop the `Layout` class invoking the `InvalidateLayout` method whenever a child is added to or removed from a layout, override the `ShouldInvalidateOnChildAdded` and `ShouldInvalidateOnChildRemoved` methods, and return `false`. The layout class can then implement a custom process when children are added or removed.

## Overriding the `OnChildMeasureInvalidated` Method

The `OnChildMeasureInvalidated` override is invoked when one of the layout's children changes size, and is shown in the following code example:

```
protected override void OnChildMeasureInvalidated()
{
    base.OnChildMeasureInvalidated();
    layoutInfoCache.Clear();
}
```

The override invalidates the child layout, and discards all of the cached layout information.

## Consuming the `WrapLayout`

The `WrapLayout` class can be consumed by placing it on a `Page` derived type, as demonstrated in the following XAML code example:

```
<ContentPage ... xmlns:local="clr-namespace:ImageWrapLayout">
    <ScrollView Margin="0,20,0,20">
        <local:WrapLayout x:Name="wrapLayout" />
    </ScrollView>
</ContentPage>
```

The equivalent C# code is shown below:

```

public class ImageWrapLayoutPageCS : ContentPage
{
    WrapLayout wrapLayout;

    public ImageWrapLayoutPageCS()
    {
        wrapLayout = new WrapLayout();

        Content = new ScrollView
        {
            Margin = new Thickness(0, 20, 0, 20),
            Content = wrapLayout
        };
    }
    ...
}

```

Children can then be added to the `WrapLayout` as required. The following code example shows `Image` elements being added to the `WrapLayout`:

```

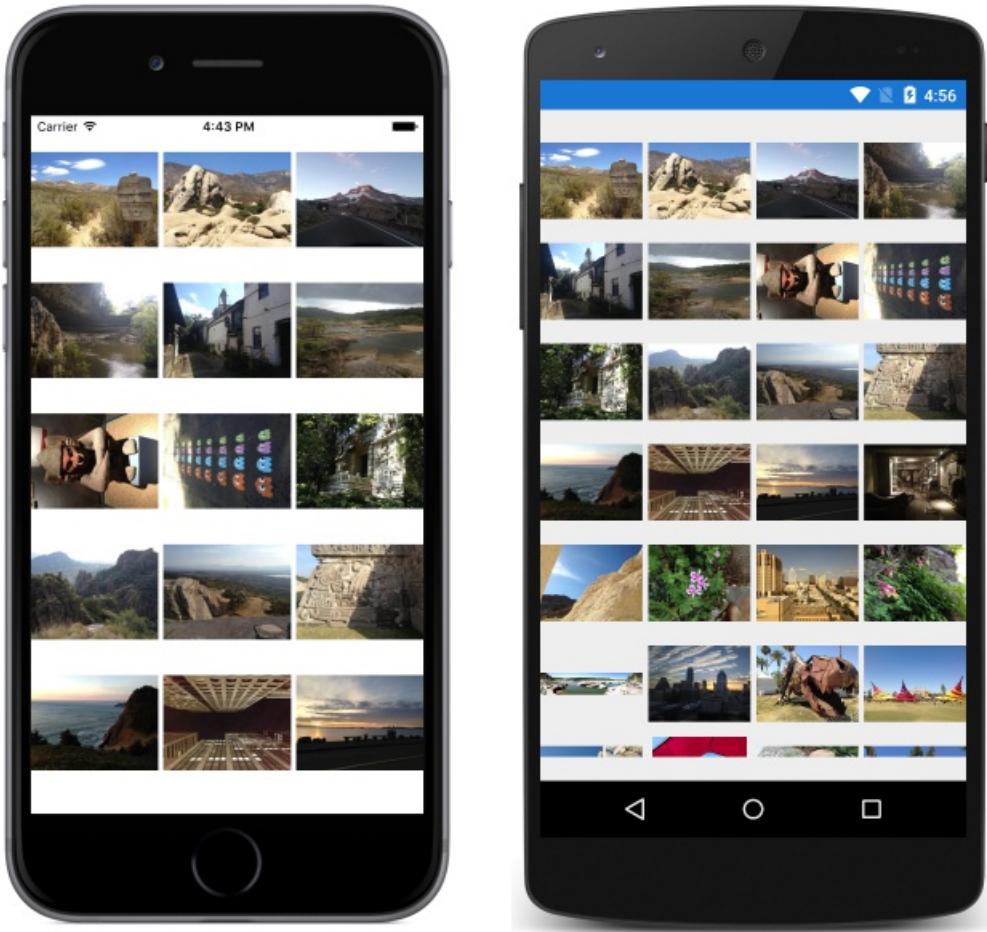
protected override async void OnAppearing()
{
    base.OnAppearing();

    var images = await GetImageListAsync();
    foreach (var photo in images.Photos)
    {
        var image = new Image
        {
            Source = ImageSource.FromUri(new Uri(photo + string.Format("?width={0}&height={0}&mode=max",
Device.RuntimePlatform == Device.UWP ? 120 : 240)))
        };
        wrapLayout.Children.Add(image);
    }
}

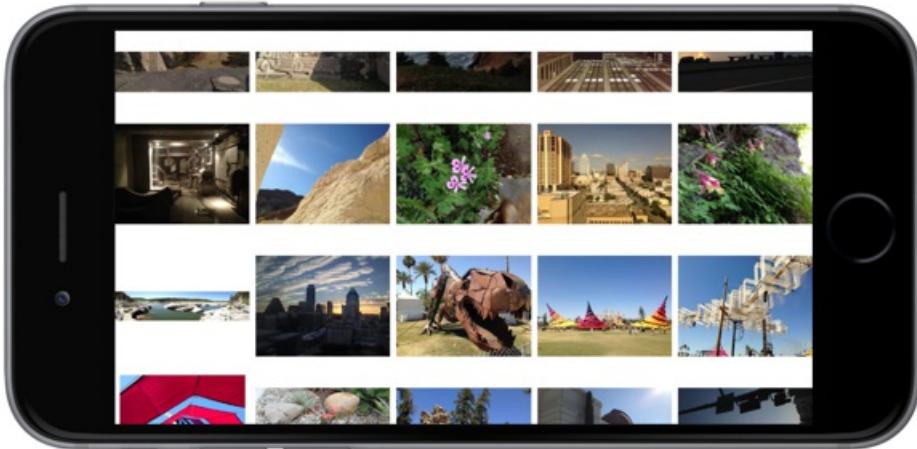
async Task<ImageList> GetImageListAsync()
{
    var requestUri = "https://docs.xamarin.com/demo/stock.json";
    using (var client = new HttpClient())
    {
        var result = await client.GetStringAsync(requestUri);
        return JsonConvert.DeserializeObject<ImageList>(result);
    }
}

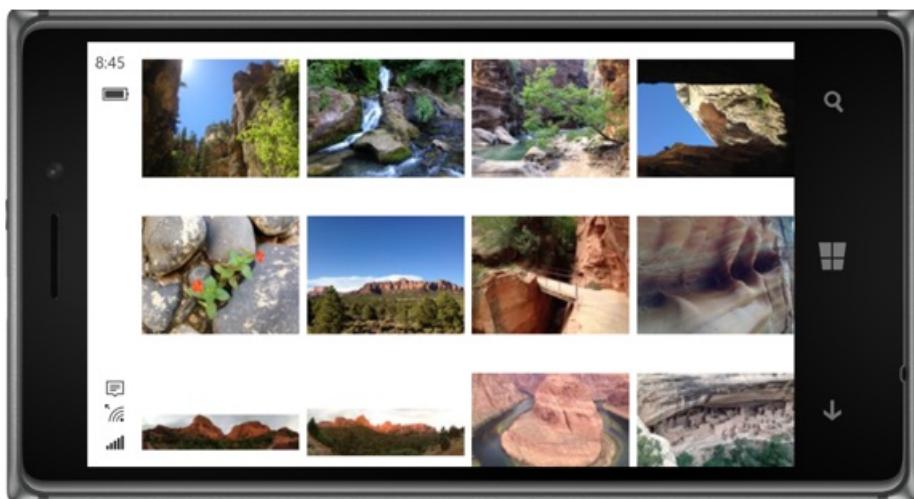
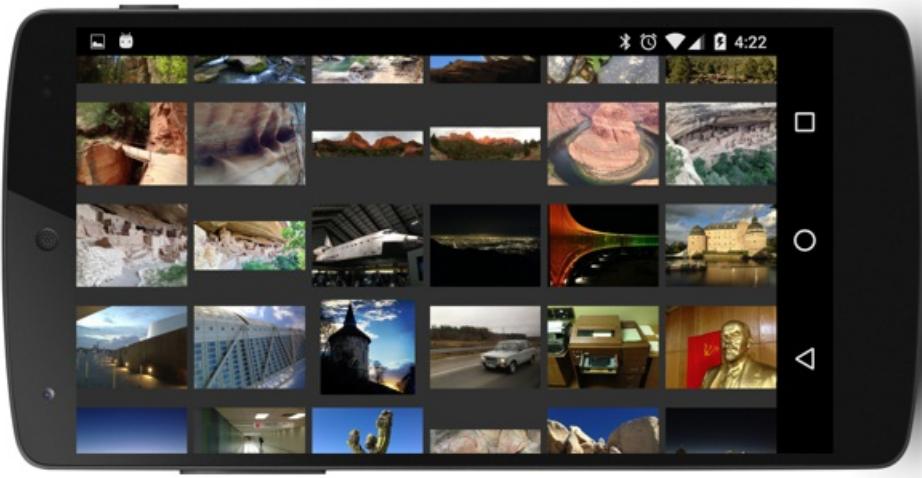
```

When the page containing the `WrapLayout` appears, the sample application asynchronously accesses a remote JSON file containing a list of photos, creates an `Image` element for each photo, and adds it to the `WrapLayout`. This results in the appearance shown in the following screenshots:



The following screenshots show the `WrapLayout` after it's been rotated to landscape orientation:





The number of columns in each row depends on the photo size, the screen width, and the number of pixels per device-independent unit. The `Image` elements asynchronously load the photos, and therefore the `WrapLayout` class will receive frequent calls to its `LayoutChildren` method as each `Image` element receives a new size based on the loaded photo.

## Summary

This article explained how to write a custom layout class, and demonstrated an orientation-sensitive `WrapLayout` class that arranges its children horizontally across the page, and then wraps the display of subsequent children to additional rows.

## Related Links

- [WrapLayout \(sample\)](#)
- [Custom Layouts](#)
- [Creating Custom Layouts in Xamarin.Forms \(video\)](#)
- [Layout](#)
- [Layout](#)
- [VisualElement](#)

# Layout Compression

7/12/2018 • 4 minutes to read • [Edit Online](#)

*Layout compression removes specified layouts from the visual tree in an attempt to improve page rendering performance. This article explains how to enable layout compression and the benefits it can bring.*

## Overview

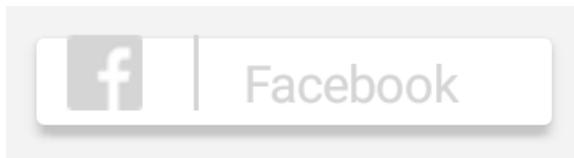
Xamarin.Forms performs layout using two series of recursive method calls:

- Layout begins at the top of the visual tree with a page, and it proceeds through all branches of the visual tree to encompass every visual element on a page. Elements that are parents to other elements are responsible for sizing and positioning their children relative to themselves.
- Invalidation is the process by which a change in an element on a page triggers a new layout cycle. Elements are considered invalid when they no longer have the correct size or position. Every element in the visual tree that has children is alerted whenever one of its children changes sizes. Therefore, a change in the size of an element in the visual tree can cause changes that ripple up the tree.

For more information about how Xamarin.Forms performs layout, see [Creating a Custom Layout](#).

The result of the layout process is a hierarchy of native controls. However, this hierarchy includes additional container renderers and wrappers for platform renderers, further inflating the view hierarchy nesting. The deeper the level of nesting, the greater the amount of work that Xamarin.Forms has to perform to display a page. For complex layouts, the view hierarchy can be both deep and broad, with multiple levels of nesting.

For example, consider the following button from the sample application for logging into Facebook:



This button is specified as a custom control with the following XAML view hierarchy:

```
<ContentView ...>
    <StackLayout>
        <StackLayout ...>
            <AbsoluteLayout ...>
                <Button ... />
                <Image ... />
                <Image ... />
                <BoxView ... />
                <Label ... />
                <Button ... />
            </AbsoluteLayout>
        </StackLayout>
        <Label ... />
    </StackLayout>
</ContentView>
```

The resulting nested view hierarchy can be examined with [Xamarin Inspector](#). On Android, the nested view hierarchy contains 17 views:

```
▼ DefaultRenderer
  ▼ DefaultRenderer
    ▼ DefaultRenderer
      ▼ DefaultRenderer
        ▼ ButtonRenderer
          AppCompatButton
        ▼ ImageRenderer
          FormslImageView
        ▼ ImageRenderer
          FormslImageView
        BoxRenderer
      ▼ LabelRenderer
        FormsTextView — "Facebook"
      ▼ ButtonRenderer
        AppCompatButton
    ▼ LabelRenderer
      FormsTextView — "Connecting..."
```

Layout compression, which is available for Xamarin.Forms applications on the iOS and Android platforms, aims to flatten the view nesting by removing specified layouts from the visual tree, which can improve page-rendering performance. The performance benefit that's delivered varies depending on the complexity of a page, the version of the operating system being used, and the device on which the application is running. However, the biggest performance gains will be seen on older devices.

#### NOTE

While this article focuses on the results of applying layout compression on Android, it's equally applicable to iOS.

## Layout Compression

In XAML, layout compression can be enabled by setting the `CompressedLayout.IsHeadless` attached property to `true` on a layout class:

```
<StackLayout CompressedLayout.IsHeadless="true">
  ...
</StackLayout>
```

Alternatively, it can be enabled in C# by specifying the layout instance as the first argument to the `CompressedLayout.SetIsHeadless` method:

```
CompressedLayout.SetIsHeadless(stackLayout, true);
```

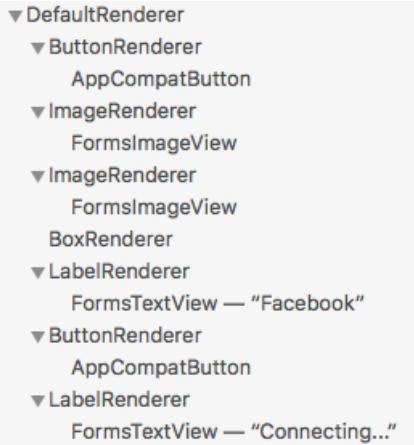
#### IMPORTANT

Since layout compression removes a layout from the visual tree, it's not suitable for layouts that have a visual appearance, or that obtain touch input. Therefore, layouts that set `VisualElement` properties (such as `BackgroundColor`, `IsVisible`, `Rotation`, `Scale`, `TranslationX` and `TranslationY`) or that accept gestures, are not candidates for layout compression. However, enabling layout compression on a layout that sets visual appearance properties, or that accepts gestures, will not result in a build or runtime error. Instead, layout compression will be applied and visual appearance properties, and gesture recognition, will silently fail.

For the Facebook button, layout compression can be enabled on the three layout classes:

```
<StackLayout CompressedLayout.IsHeadless="true">
    <StackLayout CompressedLayout.IsHeadless="true" ...>
        <AbsoluteLayout CompressedLayout.IsHeadless="true" ...>
            ...
        </AbsoluteLayout>
    </StackLayout>
    ...
</StackLayout>
```

On Android, this results in a nested view hierarchy of 14 views:



```
▼ DefaultRenderer
  ▼ ButtonRenderer
    AppCompatButton
  ▼ ImageRenderer
    FormsImage
  ▼ ImageRenderer
    FormsImage
  ▼ BoxRenderer
  ▼ LabelRenderer
    FormsTextView — "Facebook"
  ▼ ButtonRenderer
    AppCompatButton
  ▼ LabelRenderer
    FormsTextView — "Connecting..."
```

Compared to the original nested view hierarchy of 17 views, this represents a reduction in the number of views of 17%. While this reduction may appear insignificant, the view reduction over an entire page can be more significant.

## Fast Renderers

Fast renderers reduce the inflation and rendering costs of Xamarin.Forms controls on Android by flattening the resulting native view hierarchy. This further improves performance by creating fewer objects, which in turn results in a less complex visual tree and less memory use. For more information about fast renderers, see [Fast Renderers](#).

For the Facebook button in the sample application, combining layout compression and fast renderers produces a nested view hierarchy of 8 views:



```
▼ DefaultRenderer
  ButtonRenderer
  ImageRenderer
  ImageRenderer
  BoxRenderer
  LabelRenderer — "Facebook"
  ButtonRenderer
  LabelRenderer — "Connecting..."
```

Compared to the original nested view hierarchy of 17 views, this represents a reduction of 52%.

The sample application contains a page extracted from a real application. Without layout compression and fast renderers, the page produces a nested view hierarchy of 130 views on Android. Enabling fast renderers and layout compression on appropriate layout classes reduces the nested view hierarchy to 70 views, a reduction of 46%.

## Summary

Layout compression removes specified layouts from the visual tree in an attempt to improve page rendering performance. The performance benefit that this delivers varies depending on the complexity of a page, the version of the operating system being used, and the device on which the application is running. However, the biggest performance gains will be seen on older devices.

## Related Links

- [Creating a Custom Layout](#)
- [Fast Renderers](#)
- [LayoutCompression \(sample\)](#)

# Xamarin.Forms ListView

10/10/2018 • 2 minutes to read • [Edit Online](#)

ListView is a view for presenting lists of data, especially long lists that require scrolling. This guide will show you how to use ListView:

1. **Data Sources** – Populate a ListView with data, with or without data binding.
2. **Cell Appearance** – Customize the appearance of the built-in cells or create your own custom cell.
3. **List Appearance** – Customize the appearance of ListView. Set headers and footers, enable groups and change the height of rows.
4. **Interactivity** – Handle taps and selections, implement pull-to-refresh, and add contextual actions.
5. **Performance** – Avoid performance problems.

## Use Cases

Make sure ListView is the right control for your needs. ListView can be used in any situation where you are displaying scrollable lists of data. ListViews support context actions and data binding.

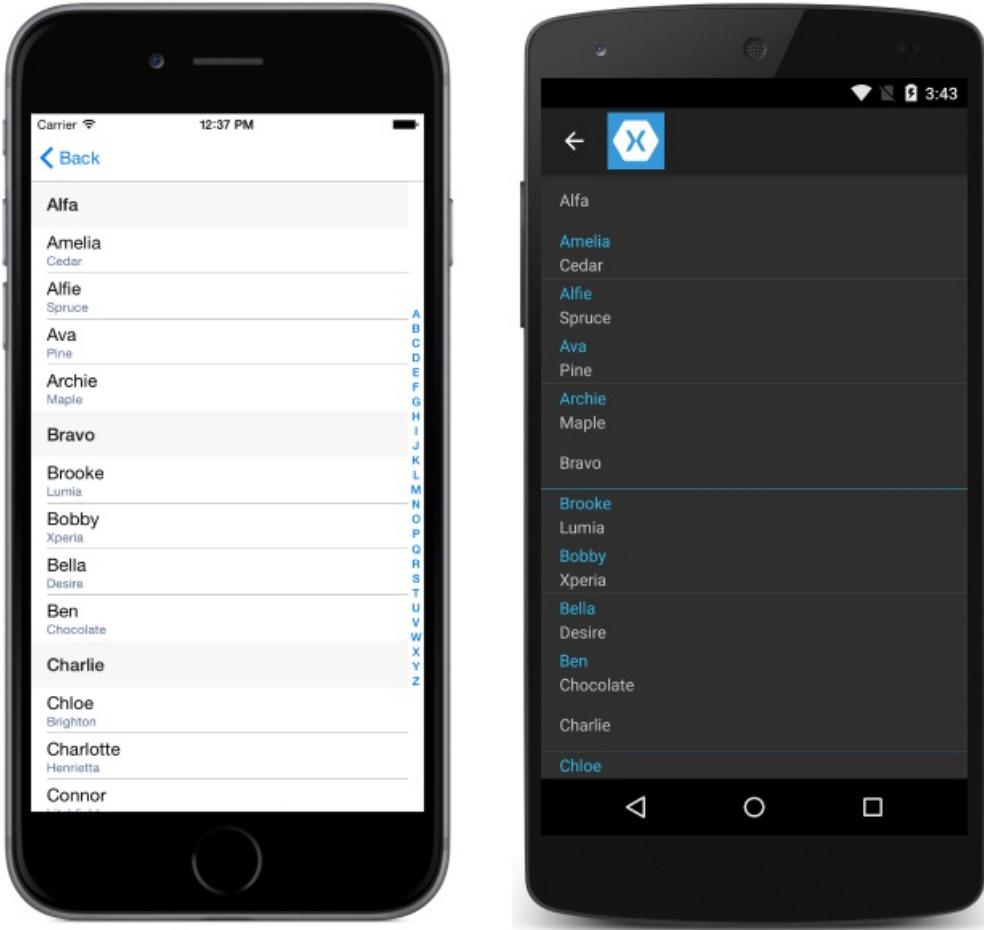
ListView should not be confused with [TableView](#). The TableView control is a better option whenever you have a non-bound list of options or data. For example, the iOS settings app, which has a mostly predefined set of options, is better suited to use TableView than ListView.

Also note that a ListView is best suited for homogeneous data – that is, all data should be of the same type. This is because only one type of cell can be used for each row in the list. TableViews can support multiple cell types, so they are a better option when you need to mix views.

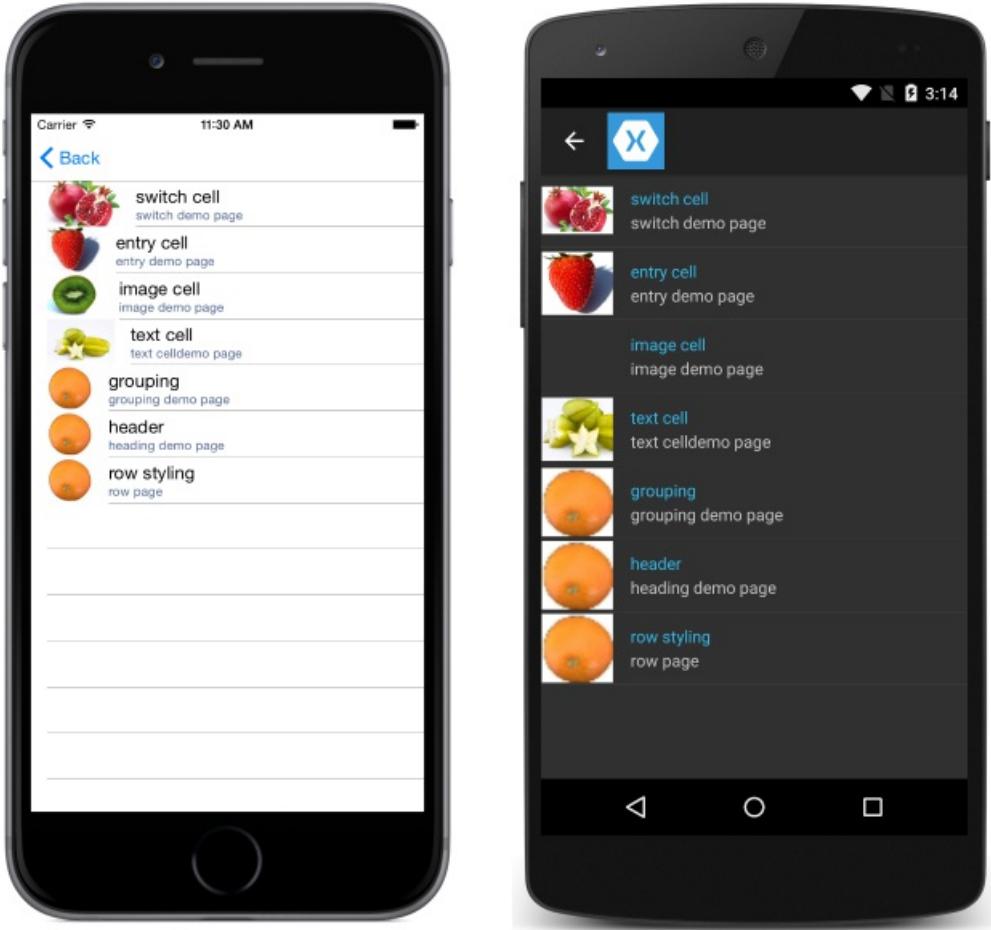
## Components

ListView has a number of components available to exercise the native functionality of each platform. Each of these components is described below:

- **Headers and Footers** – Text or view to display at the beginning and end of a list, separate from list's data. Headers and footers can be bound to a data source independently from the ListView's data source.
- **Groups** – Data in a ListView can be grouped for easier navigation. Groups are typically data bound:



- **Cells** – Data in a ListView is presented in cells. Each cell corresponds to a row of data. There are built-in cells to choose from, or you can define your own custom cell. Both built-in and custom cells can be used/defined in XAML or code.
  - **Built-in** – Built in cells, especially TextCell and ImageCell, can be great for performance, since they correspond to native controls on each platform.
    - **TextCell** – Displays a string of text, optionally with detail text. Detail text is rendered as a second line in a smaller font with an accent color.
    - **ImageCell** – Displays an image with text. Appears as a TextCell with an image on the left.
  - **Custom Cells** – Custom cells are great when you need to present complex data. For example, a custom view could be used to present a list of songs, including album and artist:

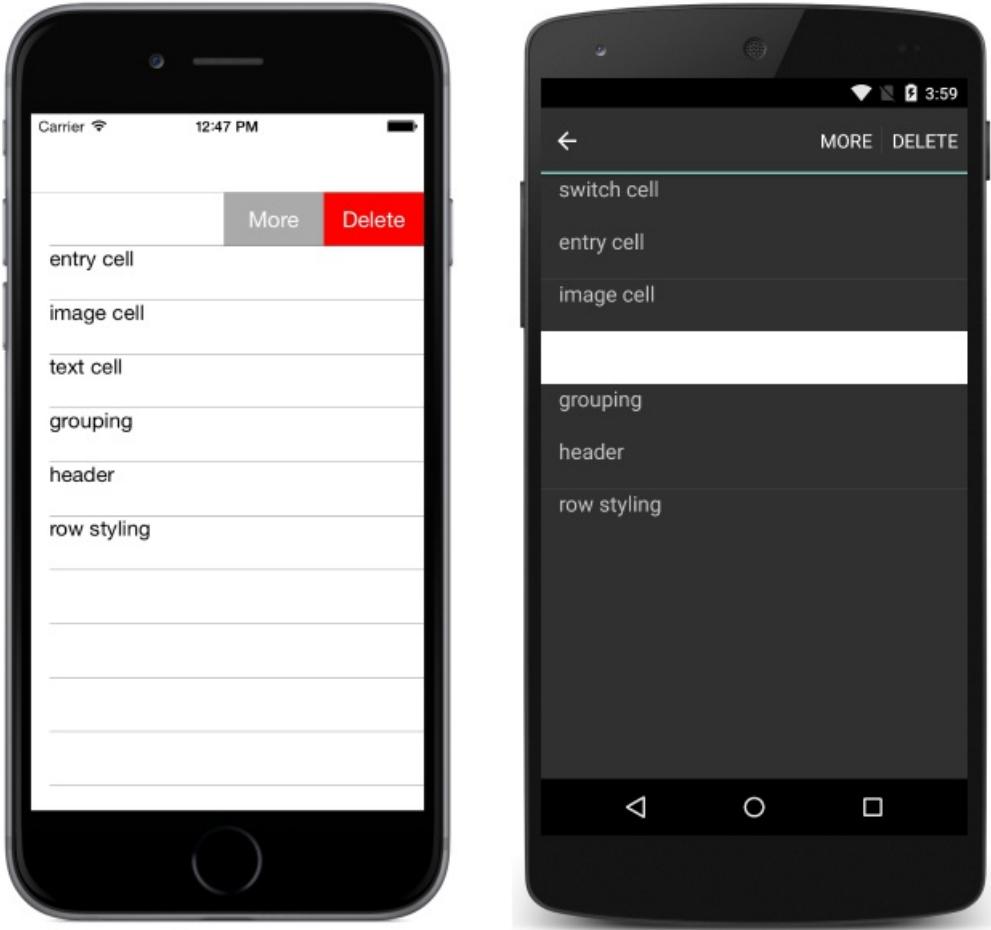


To learn more about customizing cells in a ListView, see [Customizing ListView Cell Appearance](#).

## Functionality

ListView supports a number of interaction styles, including:

- **Pull-to-Refresh** – ListView supports pull-to-refresh on each platform.
- **Context Actions** – ListView supports taking action on individual items in a list. For example, you can implement swipe-to-action on iOS, or long-tap actions on Android.
- **Selection** – You can listen for selections and deselections to take action when a row is tapped.



To learn more about the interactivity features of ListView, see [Actions & Interactivity with ListView](#).

## Related Links

- [Working With ListView \(sample\)](#)
- [Two Way Binding \(sample\)](#)
- [Built In Cells \(sample\)](#)
- [Custom Cells \(sample\)](#)
- [Grouping \(sample\)](#)
- [Custom Renderer View \(sample\)](#)
- [ListView Interactivity \(sample\)](#)

# ListView Data Sources

9/4/2018 • 3 minutes to read • [Edit Online](#)

A `ListView` is used for displaying lists of data. We'll learn about populating a ListView with data, and how we can bind to the selected item.

- **Setting ItemsSource** – uses a simple list or array.
- **Data Binding** – establishes a relationship between a model and the ListView. Binding is ideal for the MVVM pattern.

## ItemsSource

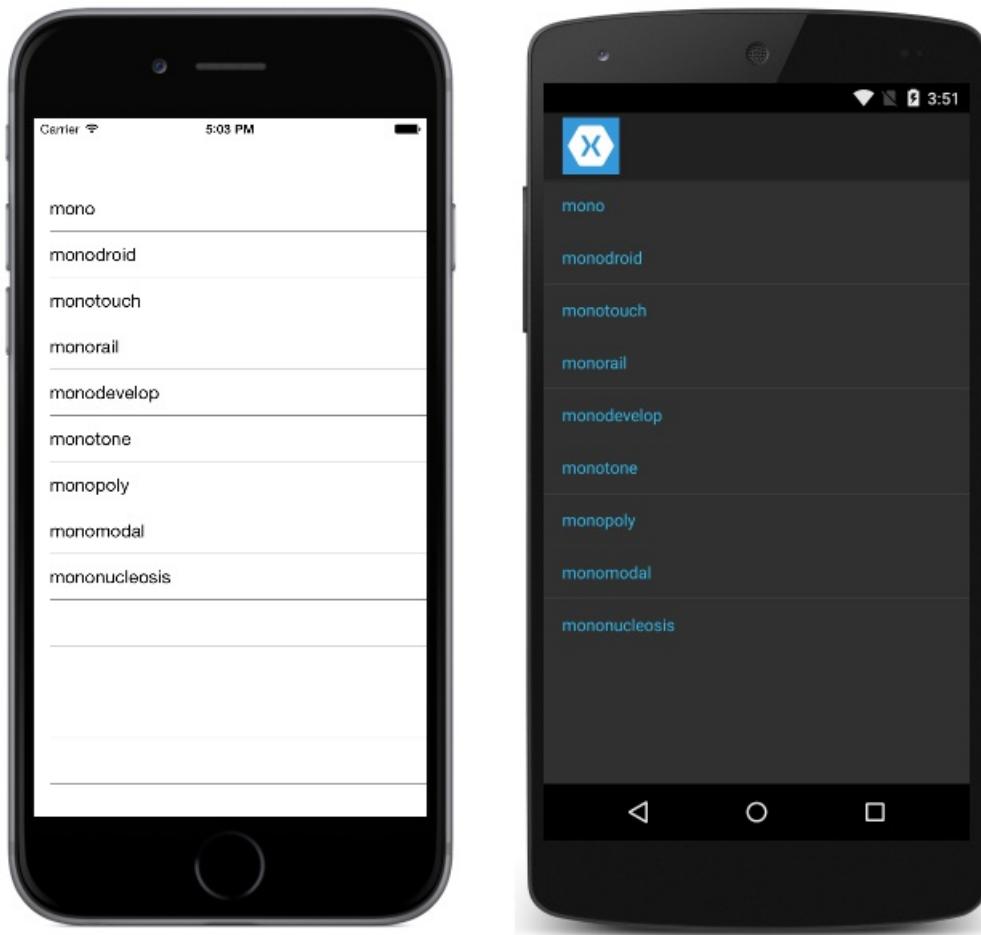
A `ListView` is populated with data using the `ItemsSource` property, which can accept any collection implementing `IEnumerable`. The simplest way to populate a `ListView` involves using an array of strings:

```
<ListView>
    <ListView.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>mono</x:String>
            <x:String>monodroid</x:String>
            <x:String>monotouch</x:String>
            <x:String>monorail</x:String>
            <x:String>monodevelop</x:String>
            <x:String>monotone</x:String>
            <x:String>monopoly</x:String>
            <x:String>monomodal</x:String>
            <x:String>mononucleosis</x:String>
        </x:Array>
    </ListView.ItemsSource>
</ListView>
```

The equivalent C# code is:

```
var listView = new ListView();
listView.ItemsSource = new string[]
{
    "mono",
    "monodroid",
    "monotouch",
    "monorail",
    "monodevelop",
    "monotone",
    "monopoly",
    "monomodal",
    "mononucleosis"
};

//monochrome will not appear in the list because it was added
//after the list was populated.
listView.ItemsSource.Add("monochrome");
```



The above approach will populate the `ListView` with a list of strings. By default, `ListView` will call `ToString` and display the result in a `TextCell` for each row. To customize how data is displayed, see [Cell Appearance](#).

Because `.ItemsSource` has been sent to an array, the content will not update as the underlying list or array changes. If you want the `ListView` to automatically update as items are added, removed and changed in the underlying list, you'll need to use an `ObservableCollection`. `ObservableCollection` is defined in `System.Collections.ObjectModel` and is just like `List`, except that it can notify `ListView` of any changes:

```
ObservableCollection<Employees> employeeList = new ObservableCollection<Employees>();
listView.ItemsSource = employeeList;

//Mr. Mono will be added to the ListView because it uses an ObservableCollection
employeeList.Add(new Employee(){ DisplayName="Mr. Mono"});
```

## Data Binding

Data binding is the "glue" that binds the properties of a user interface object to the properties of some CLR object, such as a class in your ViewModel. Data binding is useful because it simplifies the development of user interfaces by replacing a lot of boring boilerplate code.

Data binding works by keeping objects in sync as their bound values change. Instead of having to write event handlers for every time a control's value changes, you establish the binding and enable binding in your ViewModel.

For more information on data binding, see [Data Binding Basics](#) which is part four of the [Xamarin.Forms XAML Basics article series](#).

### Binding Cells

Properties of cells (and children of cells) can be bound to properties of objects in the `ItemsSource`. For example, a

`ListView` could be used to present a list of employees.

The employee class:

```
public class Employee{  
    public string DisplayName {get; set;}  
}
```

`ObservableCollection<Employee>` is created and set as the `ListView`'s `ItemsSource`:

```
ObservableCollection<Employee> employees = new ObservableCollection<Employee>();  
public EmployeeListPage()  
{  
    //defined in XAML to follow  
    EmployeeView.ItemsSource = employees;  
    ...  
}
```

The list is populated with data:

```
public EmployeeListPage()  
{  
    ...  
    employees.Add(new Employee{ DisplayName="Rob Finnerty"});  
    employees.Add(new Employee{ DisplayName="Bill Wrestler"});  
    employees.Add(new Employee{ DisplayName="Dr. Geri-Beth Hooper"});  
    employees.Add(new Employee{ DisplayName="Dr. Keith Joyce-Purdy"});  
    employees.Add(new Employee{ DisplayName="Sheri Spruce"});  
    employees.Add(new Employee{ DisplayName="Burt Indybrick"});  
}
```

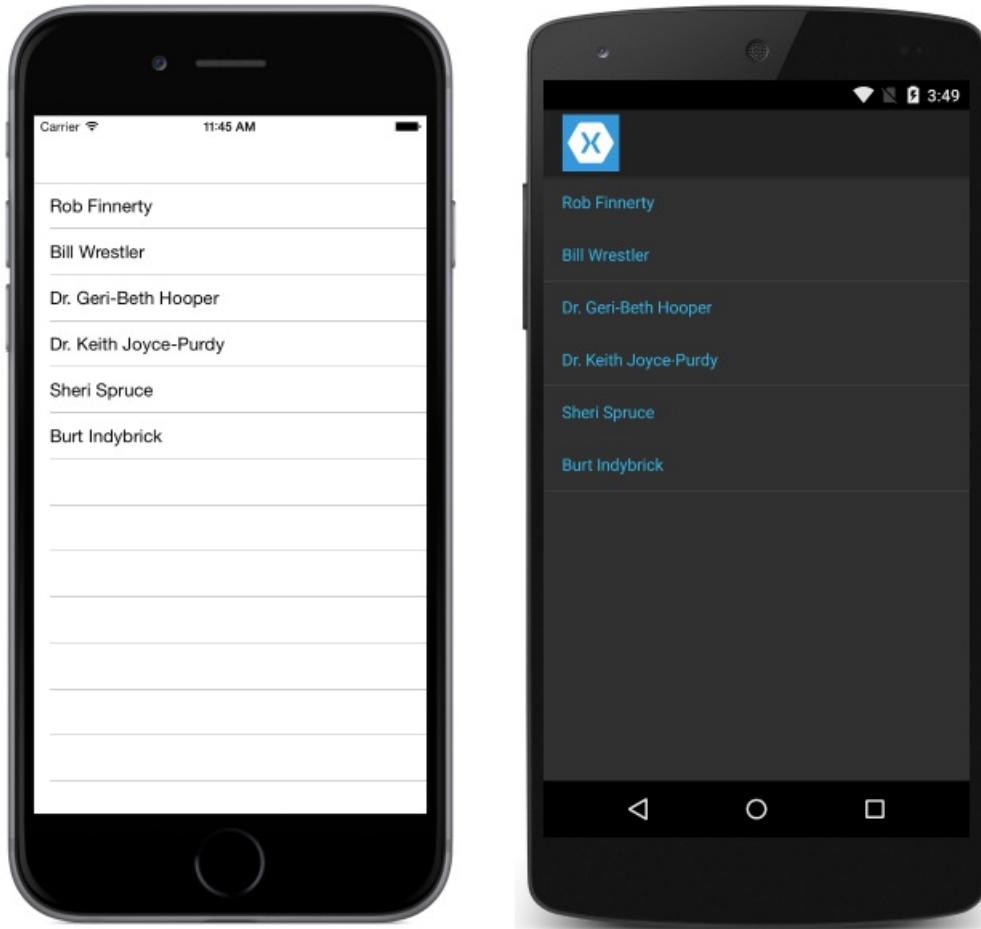
The following snippet demonstrates a `ListView` bound to a list of employees:

```
<?xml version="1.0" encoding="utf-8" ?>  
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:constants="clr-namespace:XamarinFormsSample;assembly=XamarinFormsXamlSample"  
    x:Class="XamarinFormsXamlSample.Views.EmployeeListPage"  
    Title="Employee List">  
    <ListView x:Name="EmployeeView">  
        <ListView.ItemTemplate>  
            <DataTemplate>  
                <TextCell Text="{Binding DisplayName}" />  
            </DataTemplate>  
        </ListView.ItemTemplate>  
    </ListView>  
</ContentPage>
```

Note that the binding was setup in code for simplicity, although it could have been bound in XAML.

The previous bit of XAML defines a `ContentPage` that contains a `ListView`. The data source of the `ListView` is set via the `ItemsSource` attribute. The layout of each row in the `ItemsSource` is defined within the `ListView.ItemTemplate` element.

This is the result:



## Binding SelectedItem

Often you'll want to bind to the selected item of a `ListView`, rather than use an event handler to respond to changes. To do this in XAML, bind the `SelectedItem` property:

```
<ListView x:Name="listView"
    SelectedItem="{Binding Source={x:Reference SomeLabel},
    Path=Text}">
    ...
</ListView>
```

Assuming `listView`'s `ItemsSource` is a list of strings, `SomeLabel` will have its `Text` property bound to the `SelectedItem`.

## Related Links

- [Two Way Binding \(sample\)](#)

# Customizing ListView Cell Appearance

7/12/2018 • 6 minutes to read • [Edit Online](#)

ListView presents scrollable lists, which can be customized through the use of `ViewCell`s. `ViewCells` can be used for displaying text and images, indicating a true/false state and receiving user input.

There are two approaches to getting the look you want from ListView cells:

- **Customizing built-in cells** – easier implementation and better performance at the expense of customizability.
- **Creating custom cells** – more control over the end result, but have the potential for performance issues if not implemented correctly.

## Built in Cells

Xamarin.Forms comes with built-in cells that work for many simple applications:

- **TextCell** – for displaying text
- **ImageCell** – for displaying an image with text.

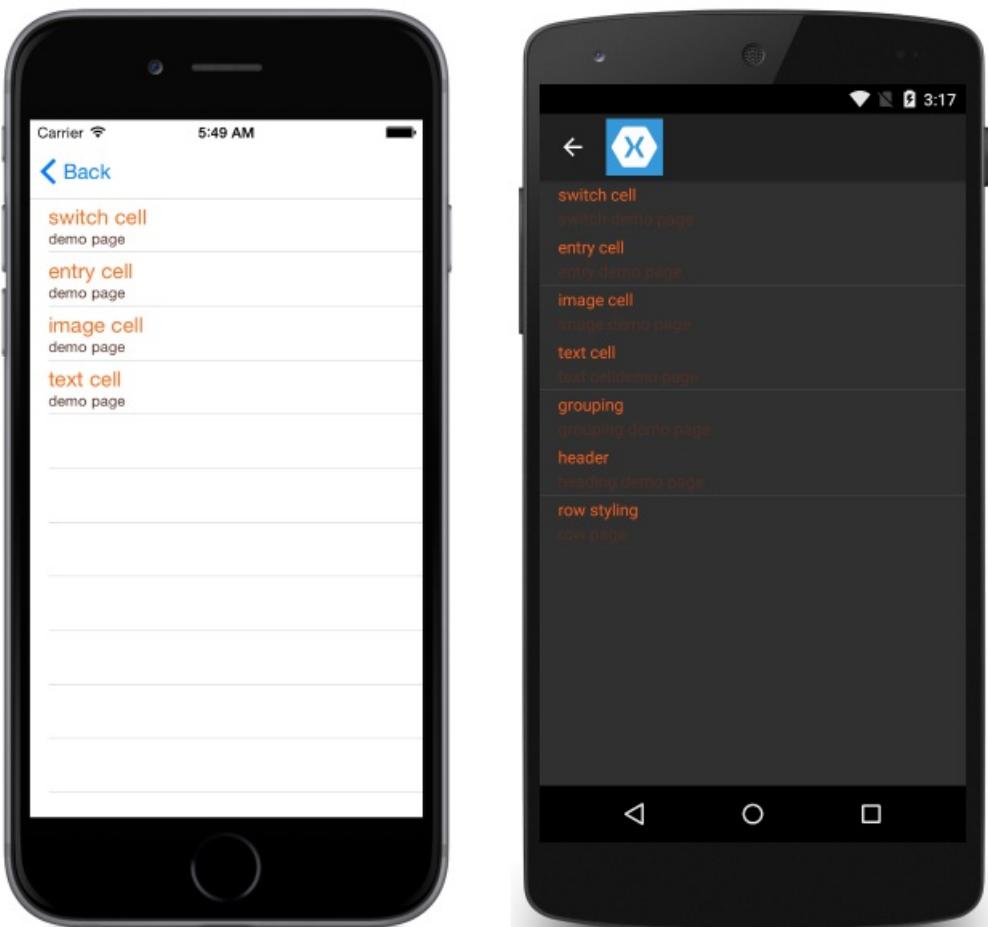
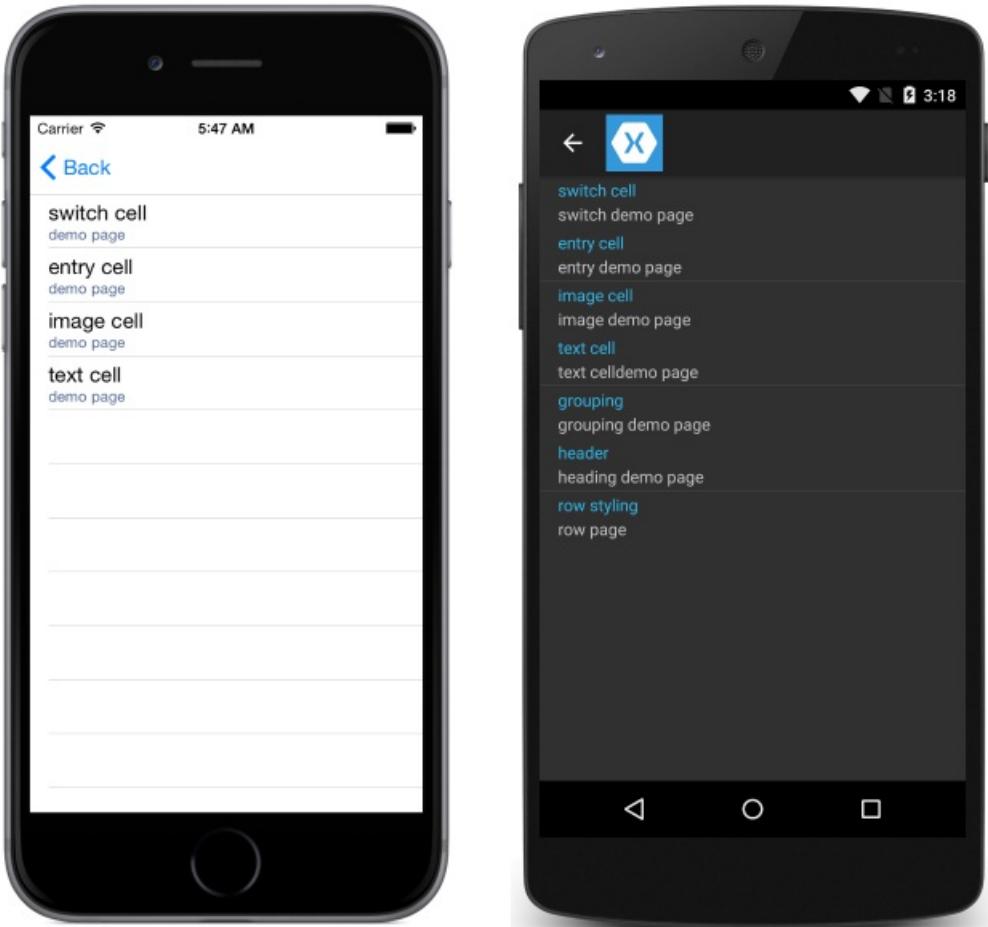
Two additional cells, `SwitchCell` and `EntryCell` are available, however they aren't commonly used with `ListView`. See `TableView` for more information about these cells.

### TextCell

`TextCell` is a cell for displaying text, optionally with a second line as detail text.

TextCells are rendered as native controls at runtime, so performance is very good compared to a custom `ViewCell`. TextCells are customizable, allowing you to set:

- `Text` – the text that is shown on the first line, in large font.
- `Detail` – the text that is shown underneath the first line, in a smaller font.
- `TextColor` – the color of the text.
- `DetailColor` – the color of the detail text



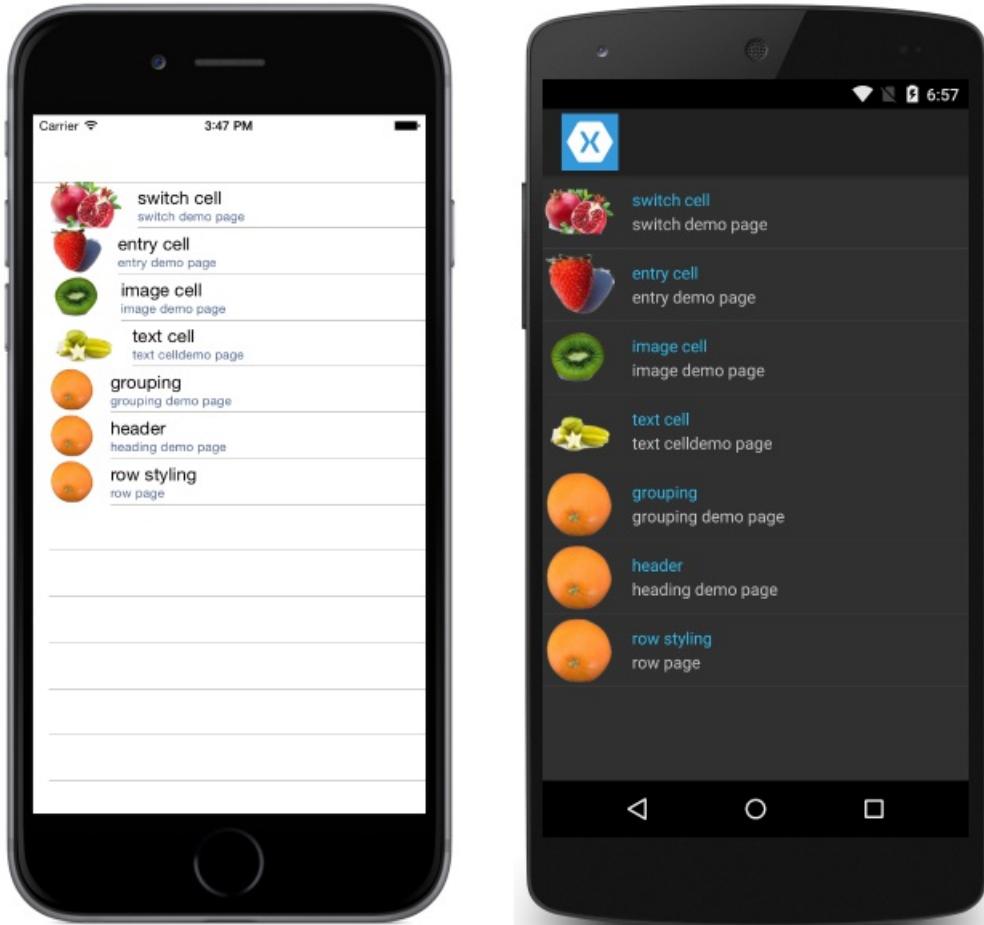
## ImageCell

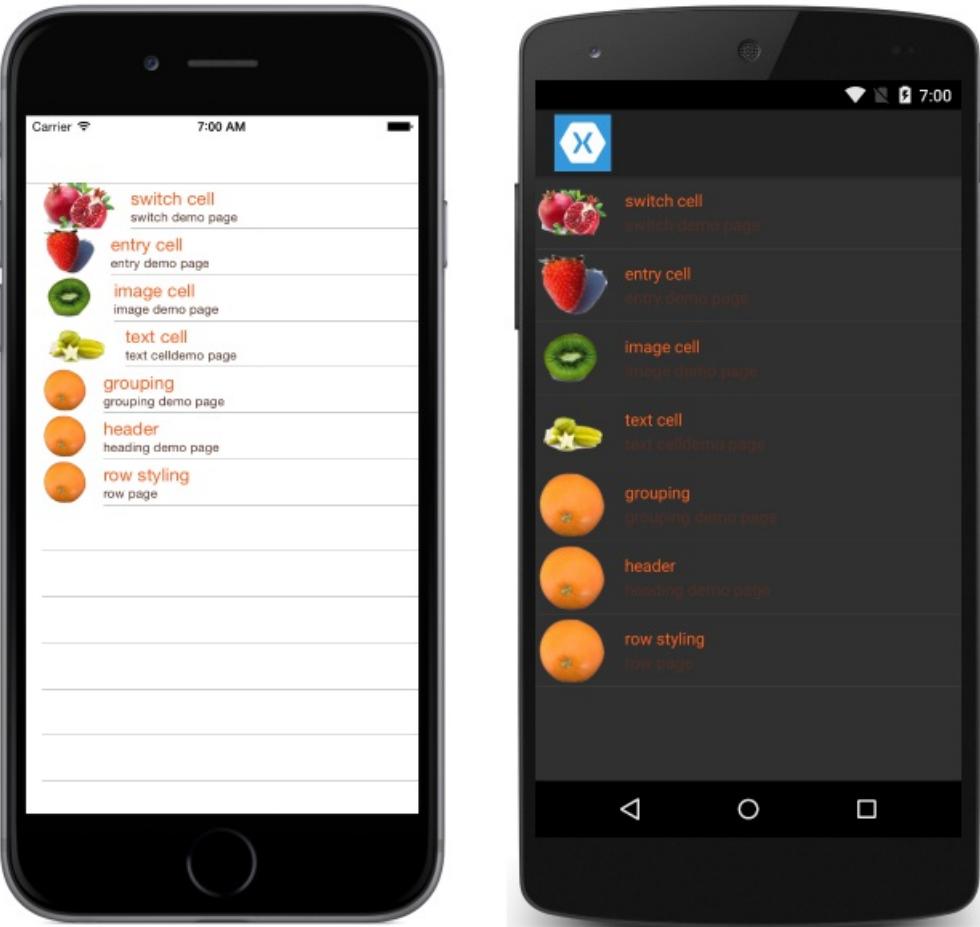
[ImageCell](#), like [TextCell](#), can be used for displaying text and secondary detail text, and it offers great

performance by using each platform's native controls. `ImageCell` differs from `TextCell` in that it displays an image to the left of the text.

`ImageCell` is useful when you need to display a list of data with a visual aspect, such as a list of contacts or movies. ImageCells are customizable, allowing you to set:

- `Text` – the text that is shown on the first line, in large font
- `Detail` – the text that is shown underneath the first line, in a smaller font
- `TextColor` – the color of the text
- `DetailColor` – the color of the detail text
- `ImageSource` – the image to display next to the text





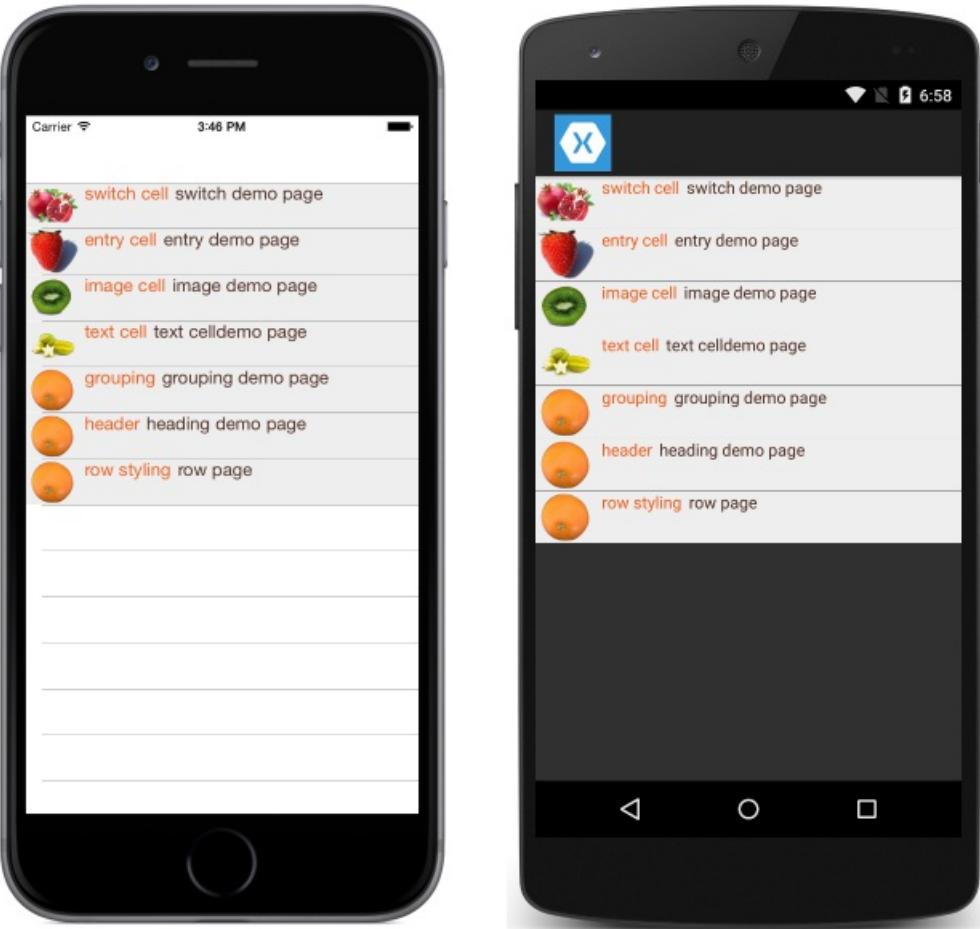
## Custom Cells

When the built-in cells don't provide the required layout, custom cells implemented the required layout. For example, you may want to present a cell with two labels that have equal weight. A `TextCell` would be insufficient because the `TextCell` has one label that is smaller. Most cell customizations add additional read-only data (such as additional labels, images or other display information).

All custom cells must derive from `ViewCell`, the same base class that all of the built-in cell types use.

Xamarin.Forms 2 introduced a new [caching behavior](#) on the `ListView` control which can be set to improve scrolling performance for some types of custom cells.

This is an example of a custom cell:



## XAML

The XAML to create the above layout is below:

```

<?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="demoListView.ImageCellPage">
    <ContentPage.Content>
        <ListView x:Name="listView">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout BackgroundColor="#eee"
                            Orientation="Vertical">
                            <StackLayout Orientation="Horizontal">
                                <Image Source="{Binding image}" />
                                <Label Text="{Binding title}"
                                    TextColor="#f35e20" />
                                <Label Text="{Binding subtitle}"
                                    HorizontalOptions="EndAndExpand"
                                    TextColor="#503026" />
                            </StackLayout>
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

The XAML above is doing a lot. Let's break it down:

- The custom cell is nested inside a `DataTemplate`, which is inside `ListView.ItemTemplate`. This is the same

process as using any other cell.

- `ViewCell` is the type of the custom cell. The child of the `DataTemplate` element must be of or derive from type `ViewCell`.
- Notice that inside the `ViewCell`, layout is managed by a `StackLayout`. This layout allows us to customize the background color. Note that any property of `StackLayout` that is bindable can be bound inside a custom cell, although that is not shown here.

## C#

Specifying a custom cell in C# is a bit more verbose than the XAML equivalent. Let's take a look:

First, define a custom cell class, with `ViewCell` as the base class:

```
public class CustomCell : ViewCell
{
    public CustomCell()
    {
        //instantiate each of our views
        var image = new Image ();
        StackLayout cellWrapper = new StackLayout ();
        StackLayout horizontalLayout = new StackLayout ();
        Label left = new Label ();
        Label right = new Label ();

        //set bindings
        left.SetBinding (Label.TextProperty, "title");
        right.SetBinding (Label.TextProperty, "subtitle");
        image.SetBinding (Image.SourceProperty, "image");

        //Set properties for desired design
        cellWrapper.BackgroundColor = Color.FromHex ("#eee");
        horizontalLayout.Orientation = StackOrientation.Horizontal;
        right.HorizontalOptions = LayoutOptions.EndAndExpand;
        left.TextColor = Color.FromHex ("#f35e20");
        right.TextColor = Color.FromHex ("503026");

        //add views to the view hierarchy
        horizontalLayout.Children.Add (image);
        horizontalLayout.Children.Add (left);
        horizontalLayout.Children.Add (right);
        cellWrapper.Children.Add (horizontalLayout);
        View = cellWrapper;
    }
}
```

In your constructor for the page with the `ListView`, set the `ListView`'s `ItemTemplate` property to a new `DataTemplate`:

```
public partial class ImageCellPage : ContentPage
{
    public ImageCellPage ()
    {
        InitializeComponent ();
        listView.ItemTemplate = new DataTemplate (typeof(CustomCell));
    }
}
```

Note that the constructor for `DataTemplate` takes a type. The `typeof` operator gets the CLR type for `CustomCell`.

## Binding Context Changes

When binding to a custom cell type's `BindableProperty` instances, the UI controls displaying the

`BindableProperty` values should use the `OnBindingContextChanged` override to set the data to be displayed in each cell, rather than the cell constructor, as demonstrated in the following code example:

```
public class CustomCell : ViewCell
{
    Label nameLabel, ageLabel, locationLabel;

    public static readonly BindableProperty NameProperty =
        BindableProperty.Create ("Name", typeof(string), typeof(CustomCell), "Name");
    public static readonly BindableProperty AgeProperty =
        BindableProperty.Create ("Age", typeof(int), typeof(CustomCell), 0);
    public static readonly BindableProperty LocationProperty =
        BindableProperty.Create ("Location", typeof(string), typeof(CustomCell), "Location");

    public string Name {
        get { return(string)GetValue (NameProperty); }
        set { SetValue (NameProperty, value); }
    }

    public int Age {
        get { return(int)GetValue (AgeProperty); }
        set { SetValue (AgeProperty, value); }
    }

    public string Location {
        get { return(string)GetValue (LocationProperty); }
        set { SetValue (LocationProperty, value); }
    }
    ...

    protected override void OnBindingContextChanged ()
    {
        base.OnBindingContextChanged ();

        if (BindingContext != null) {
            nameLabel.Text = Name;
            ageLabel.Text = Age.ToString ();
            locationLabel.Text = Location;
        }
    }
}
```

The `OnBindingContextChanged` override will be called when the `BindingContextChanged` event fires, in response to the value of the `BindingContext` property changing. Therefore, when the `BindingContext` changes, the UI controls displaying the `BindableProperty` values should set their data. Note that the `BindingContext` should be checked for a `null` value, as this can be set by Xamarin.Forms for garbage collection, which in turn will result in the `OnBindingContextChanged` override being called.

Alternatively, UI controls can bind to the `BindableProperty` instances to display their values, which removes the need to override the `OnBindingContextChanged` method.

#### NOTE

When overriding `OnBindingContextChanged`, ensure that the base class's `OnBindingContextChanged` method is called so that registered delegates receive the `BindingContextChanged` event.

In XAML, binding the custom cell type to data can be achieved as shown in the following code example:

```
<ListView x:Name="listView">
    <ListView.ItemTemplate>
        <DataTemplate>
            <local:CustomCell Name="{Binding Name}" Age="{Binding Age}" Location="{Binding Location}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

This binds the `Name`, `Age`, and `Location` bindable properties in the `CustomCell` instance, to the `Name`, `Age`, and `Location` properties of each object in the underlying collection.

The equivalent binding in C# is shown in the following code example:

```
var customCell = new DataTemplate (typeof(CustomCell));
customCell.SetBinding (CustomCell.NameProperty, "Name");
customCell.SetBinding (CustomCell.AgeProperty, "Age");
customCell.SetBinding (CustomCell.LocationProperty, "Location");

var listView = new ListView {
    ItemsSource = people,
    ItemTemplate = customCell
};
```

On iOS and Android, if the `ListView` is recycling elements and the custom cell uses a custom renderer, the custom renderer must correctly implement property change notification. When cells are reused their property values will change when the binding context is updated to that of an available cell, with `PropertyChanged` events being raised. For more information, see [Customizing a ViewCell](#). For more information about cell recycling, see [Caching Strategy](#).

## Related Links

- [Built in Cells \(sample\)](#)
- [Custom Cells \(sample\)](#)
- [Binding Context Changed \(sample\)](#)

# Customizing ListView Appearance

7/12/2018 • 5 minutes to read • [Edit Online](#)

`ListView` has options for controlling the presentation of the overall list, in addition to the underlying `ViewCell`s.

Options include:

- **Grouping** – group items in ListView for easier navigation and improved organization.
- **Headers and Footers** – display information at the beginning and end of the view that scrolls with the other items.
- **Row Separators** – show or hide separator lines between items.
- **Variable Height Rows** – by default all rows are the same height, but this can be changed to allow rows with differing heights to be displayed.

## Grouping

Often, large sets of data can become unwieldy when presented in a continuously scrolling list. Enabling grouping can improve the user experience in these cases by better organizing the content and activating platform-specific controls that make navigating data easier.

When grouping is activated for a `ListView`, a header row is added for each group.

To enable grouping:

- Create a list of lists (a list of groups, each group being a list of elements).
- Set the `ListView`'s `ItemsSource` to that list.
- Set `IsGroupingEnabled` to true.
- Set `GroupDisplayBinding` to bind to the property of the groups that is being used as the title of the group.
- [Optional] Set `GroupShortNameBinding` to bind to the property of the groups that is being used as the short name for the group. The short name is used for the jump lists (right-side column on iOS).

Start by creating a class for the groups:

```
public class PageTypeGroup : List<PageModel>
{
    public string Title { get; set; }
    public string ShortName { get; set; } //will be used for jump lists
    public string Subtitle { get; set; }
    private PageTypeGroup(string title, string shortName)
    {
        Title = title;
        ShortName = shortName;
    }

    public static IList<PageTypeGroup> All { private set; get; }
}
```

In the above code, `All` is the list that will be given to our `ListView` as the binding source. `Title` and `ShortName` are the properties that will be used for group headings.

At this stage, `All` is an empty list. Add a static constructor so that the list will be populated at program start:

```

static PageTypeGroup()
{
    List<PageTypeGroup> Groups = new List<PageTypeGroup> {
        new PageTypeGroup ("Alfa", "A"){
            new PageModel("Amelia", "Cedar", new switchCellPage(),""),
            new PageModel("Alfie", "Spruce", new switchCellPage(), "grapefruit.jpg"),
            new PageModel("Ava", "Pine", new switchCellPage(), "grapefruit.jpg"),
            new PageModel("Archie", "Maple", new switchCellPage(), "grapefruit.jpg")
        },
        new PageTypeGroup ("Bravo", "B"){
            new PageModel("Brooke", "Lumia", new switchCellPage(),""),
            new PageModel("Bobby", "Xperia", new switchCellPage(), "grapefruit.jpg"),
            new PageModel("Bella", "Desire", new switchCellPage(), "grapefruit.jpg"),
            new PageModel("Ben", "Chocolate", new switchCellPage(), "grapefruit.jpg")
        }
    }
    All = Groups; //set the publicly accessible list
}

```

In the above code we can also call `Add` on elements of `Groups`, which are instances of type `PageTypeGroup`. This is possible because `PageTypeGroup` inherits from `List<PageModel>`. This is an example of the list of lists pattern noted above.

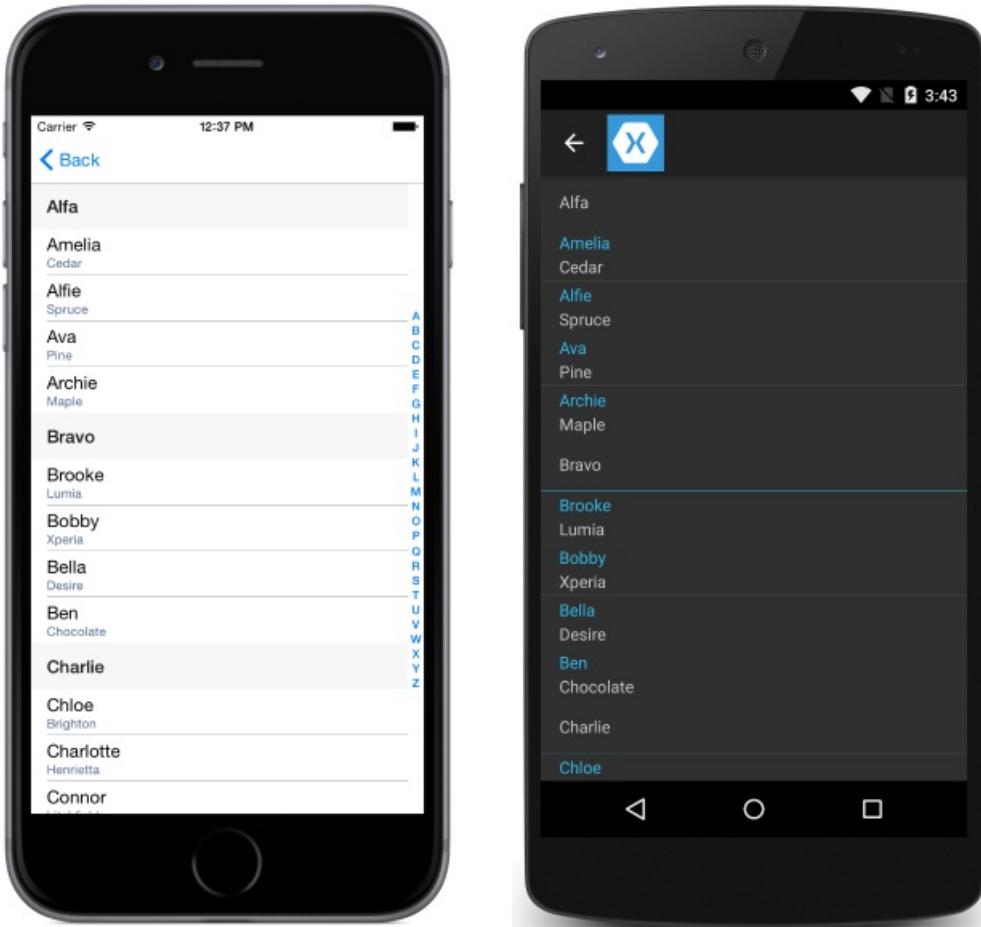
Here is the XAML for displaying the grouped list:

```

<?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="DemoListView.GroupingViewPage"
    <ContentPage.Content>
        <ListView x:Name="GroupedView"
            GroupDisplayBinding="{Binding Title}"
            GroupShortNameBinding="{Binding ShortName}"
            IsGroupingEnabled="true">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                        Detail="{Binding Subtitle}" />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </ContentPage.Content>
</ContentPage>

```

This results in the following:



Note that we have:

- Set `GroupShortNameBinding` to the `ShortName` property defined in our group class
- Set `GroupDisplayBinding` to the `Title` property defined in our group class
- Set `IsGroupingEnabled` to true
- Changed the `ListView`'s `ItemsSource` to the grouped list

### Customizing Grouping

If grouping has been enabled in the list, the group header can also be customized.

Similar to how the `ListView` has an `ItemTemplate` for defining how rows are displayed, `ListView` has a `GroupHeaderTemplate`.

An example of customizing the group header in XAML is shown here:

```

<?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="DemoListView.GroupingViewPage">
    <ContentPage.Content>
        <ListView x:Name="GroupedView"
            GroupDisplayBinding="{Binding Title}"
            GroupShortNameBinding="{Binding ShortName}"
            IsGroupingEnabled="true">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                        Detail="{Binding Subtitle}"
                        TextColor="#f35e20"
                        DetailColor="#503026" />
                </DataTemplate>
            </ListView.ItemTemplate>
            <!-- Group Header Customization-->
            <ListView.GroupHeaderTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                        Detail="{Binding ShortName}"
                        TextColor="#f35e20"
                        DetailColor="#503026" />
                </DataTemplate>
            </ListView.GroupHeaderTemplate>
            <!-- End Group Header Customization -->
        </ListView>
    </ContentPage.Content>
</ContentPage>

```

## Headers and Footers

It is possible for a ListView to present a header and footer that scroll with the elements of the list. The header and footer can be strings of text or a more complicated layout. Note that this is separate from [section groups](#).

You can set the `Header` and/or `Footer` to a simple string value, or you can set them to a more complex layout. There are also `HeaderTemplate` and `FooterTemplate` properties that let you create more complex layouts for the header and footer that support data binding.

To create a simple header/footer, just set the Header or Footer properties to the text you want to display. In code:

```

ListView HeaderList = new ListView() {
    Header = "Header",
    Footer = "Footer"
};

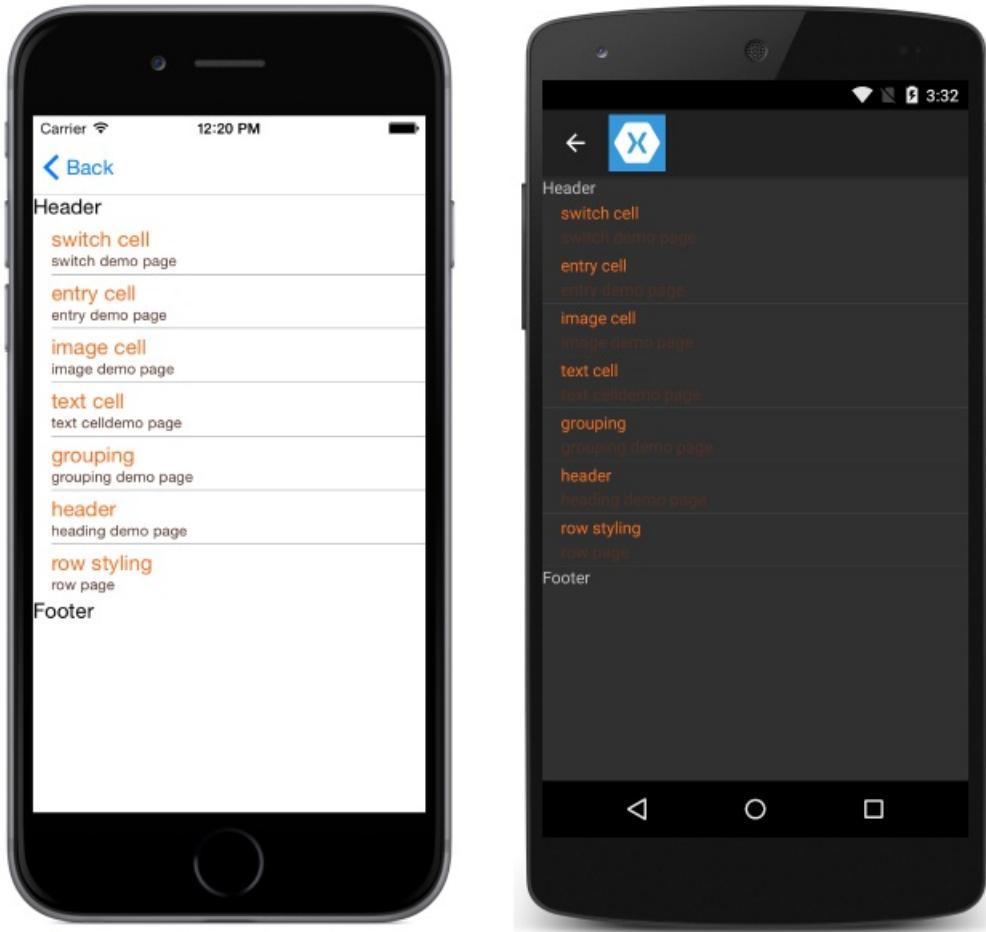
```

In XAML:

```

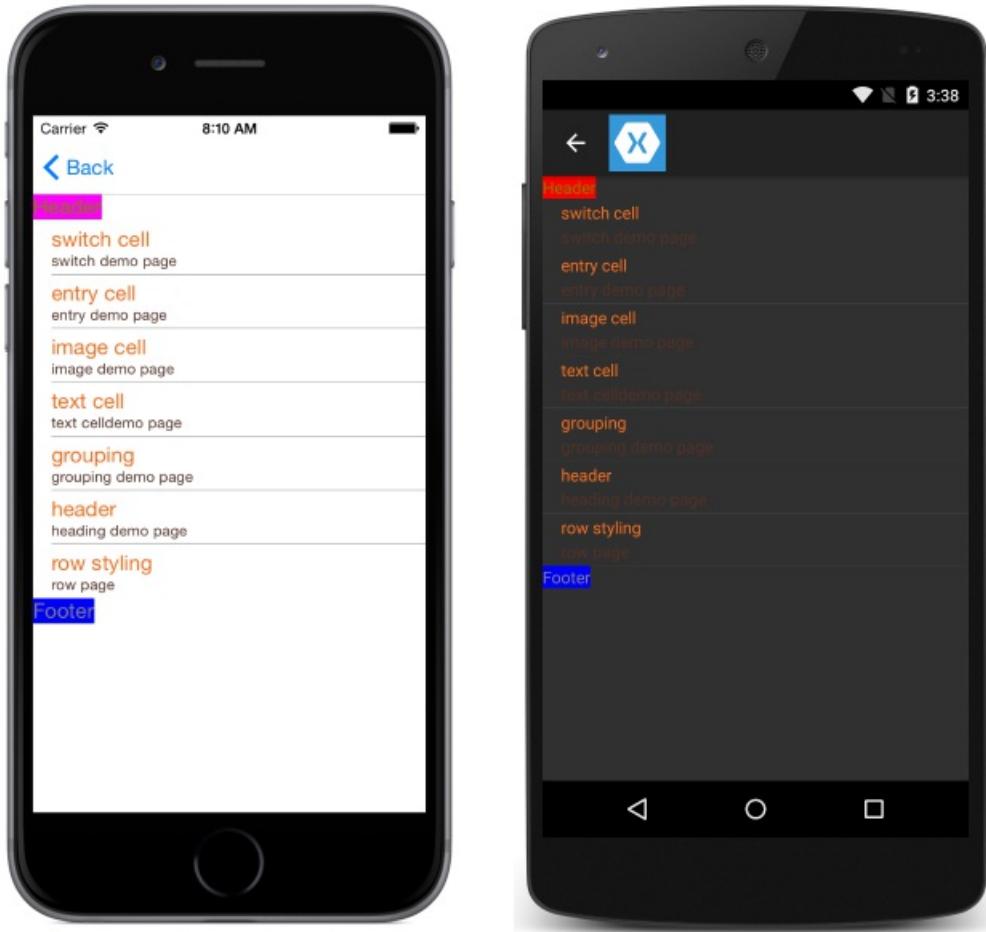
<ListView x:Name="HeaderList" Header="Header" Footer="Footer"></ListView>

```



To create a customized header and footer, define the Header and Footer views:

```
<ListView.Header>
    <StackLayout Orientation="Horizontal">
        <Label Text="Header"
              TextColor="Olive"
              BackgroundColor="Red" />
    </StackLayout>
</ListView.Header>
<ListView.Footer>
    <StackLayout Orientation="Horizontal">
        <Label Text="Footer"
              TextColor="Gray"
              BackgroundColor="Blue" />
    </StackLayout>
</ListView.Footer>
```



## Row Separators

Separator lines are displayed between `ListView` elements by default on iOS and Android. If you'd prefer to hide the separator lines on iOS and Android, set the `SeparatorVisibility` property on your `ListView`. The options for `SeparatorVisibility` are:

- **Default** - shows a separator line on iOS and Android.
- **None** - hides the separator on all platforms.

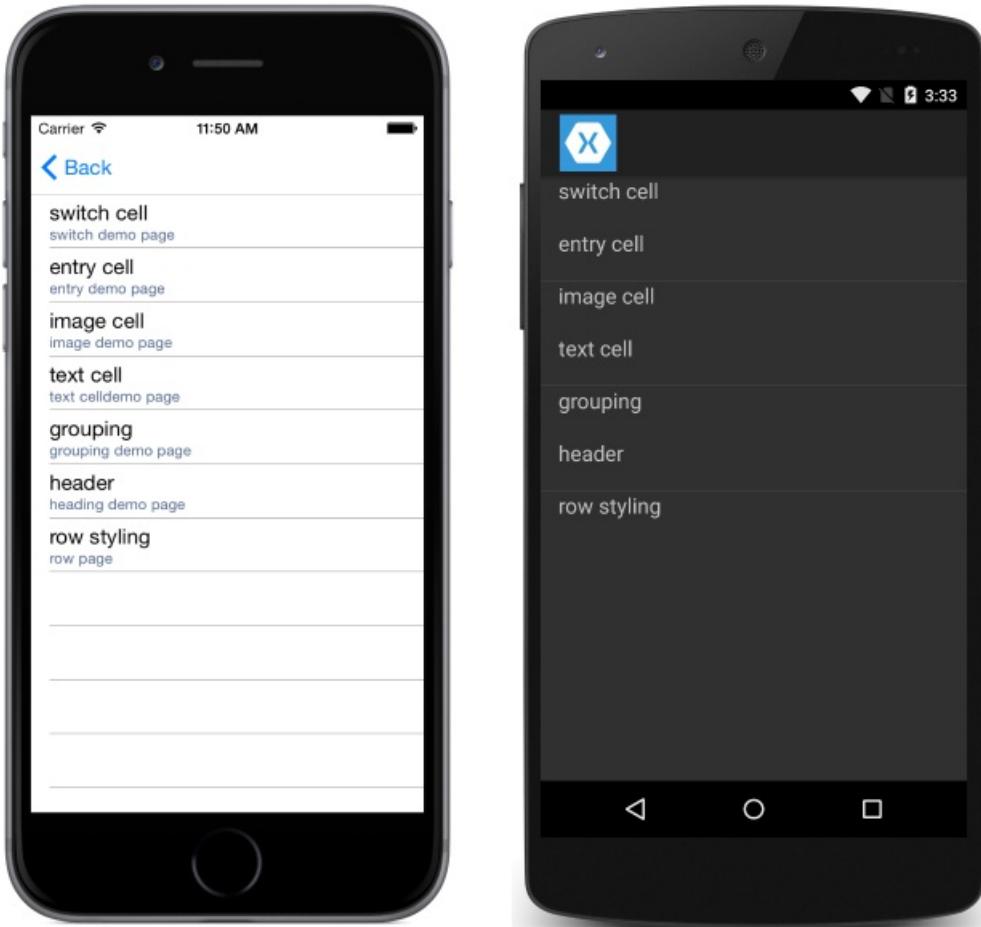
Default Visibility:

C#:

```
SepratorDemoListView.SeparatorVisibility = SeparatorVisibility.Default;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="Default" />
```



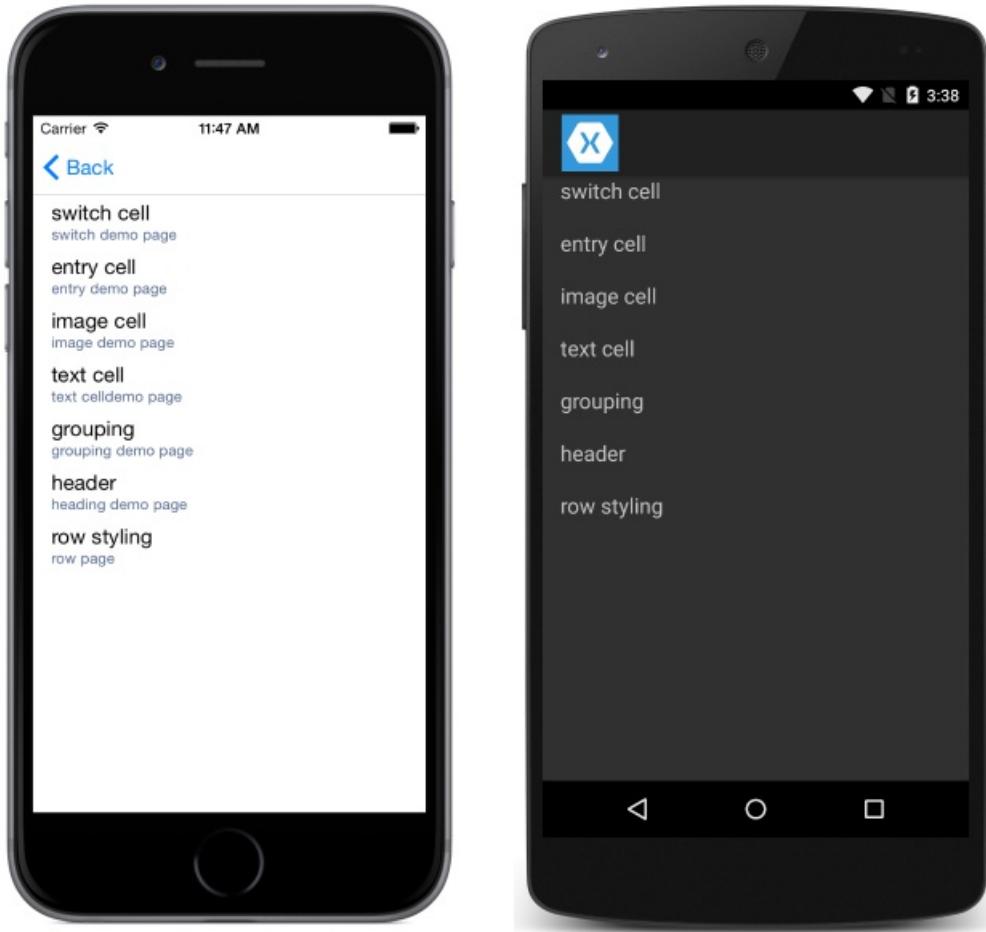
None:

C#:

```
SepratorDemoListView.SeparatorVisibility = SeparatorVisibility.None;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="None" />
```



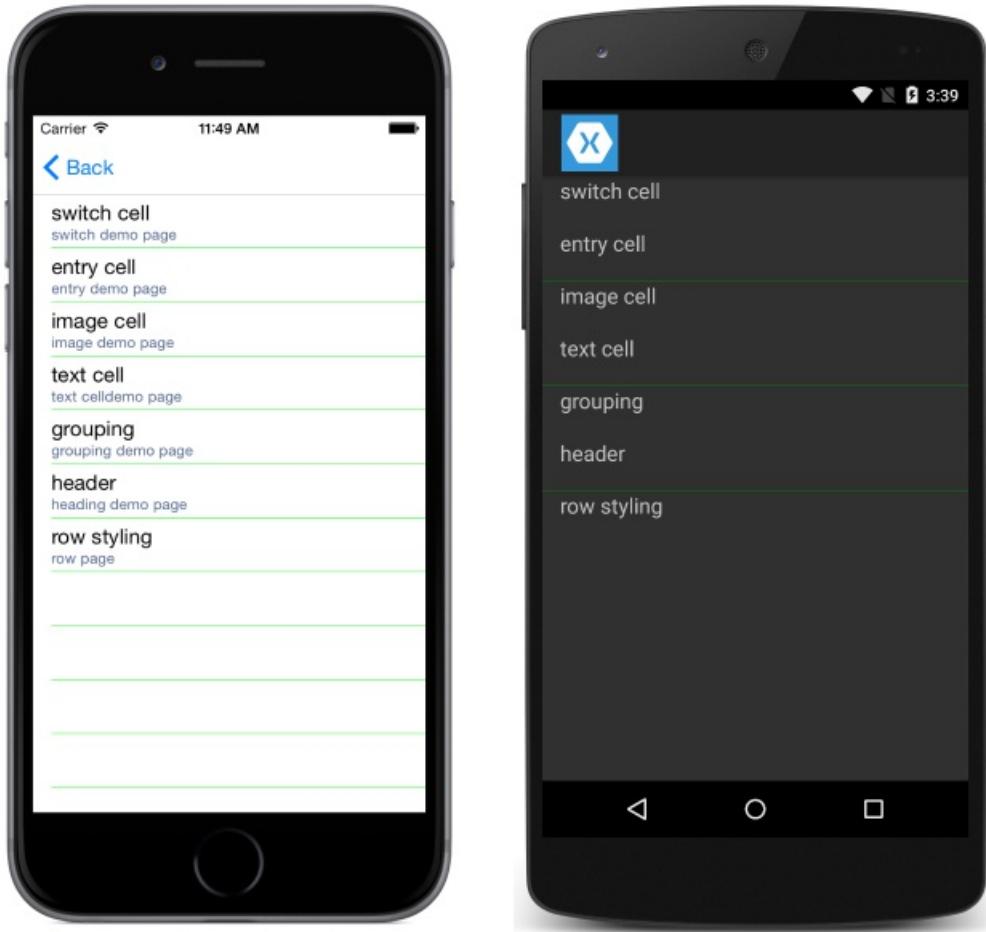
You can also set the color of the separator line via the `SeparatorColor` property:

C#:

```
SepratorDemoListView.SeparatorColor = Color.Green;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorColor="Green" />
```



#### NOTE

Setting either of these properties on Android after loading the `ListView` incurs a large performance penalty.

## Row Heights

All rows in a ListView have the same height by default. ListView has two properties that can be used to change that behavior:

- `HasUnevenRows` – `true` / `false` value, rows have varying heights if set to `true`. Defaults to `false`.
- `RowHeight` – sets the height of each row when `HasUnevenRows` is `false`.

You can set the height of all rows by setting the `RowHeight` property on the `ListView`.

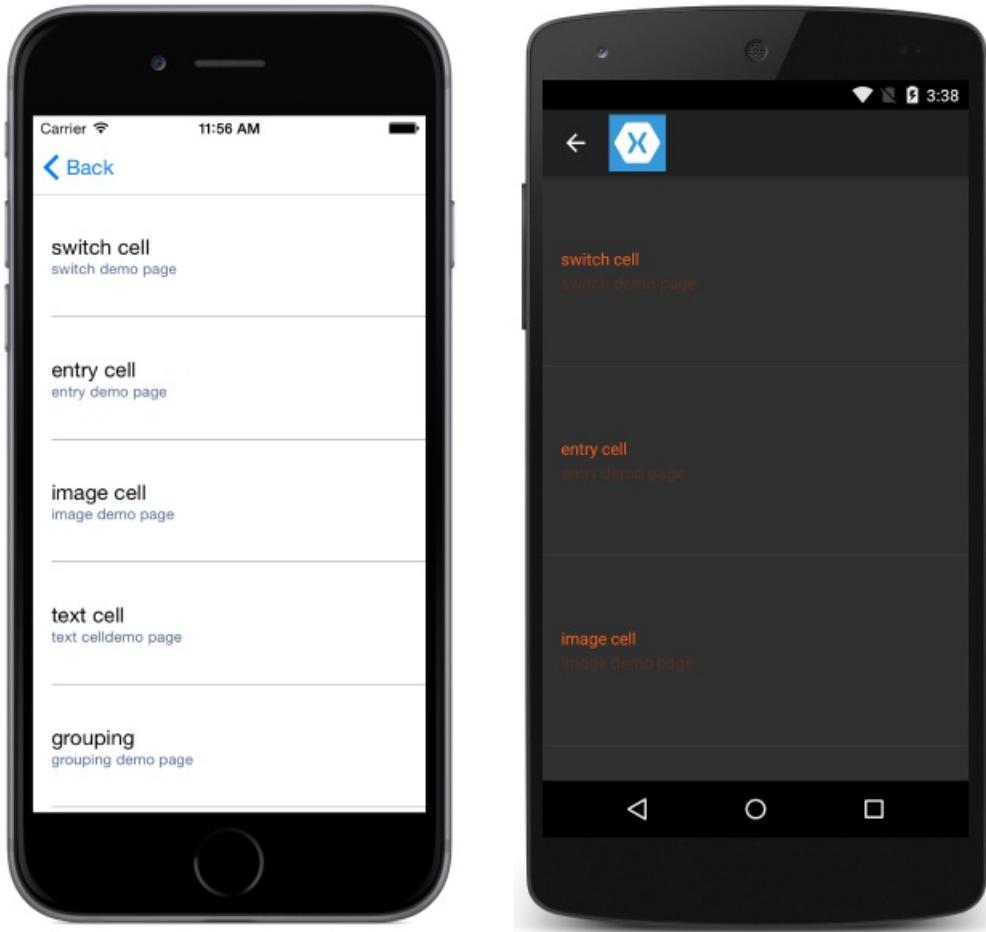
### Custom Fixed Row Height

C#:

```
RowHeightDemoListView.RowHeight = 100;
```

XAML:

```
<ListView x:Name="RowHeightDemoListView" RowHeight="100" />
```



## Uneven Rows

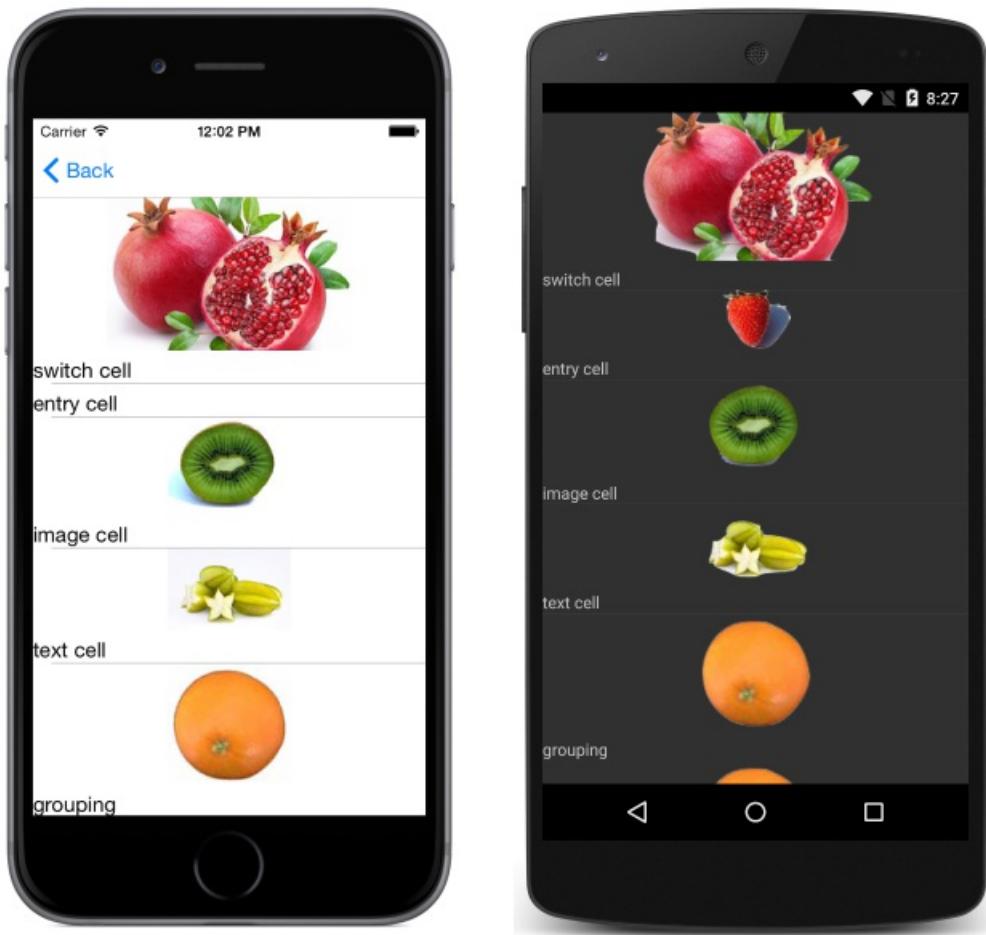
If you'd like individual rows to have different heights, you can set the `HasUnevenRows` property to `true`. Note that row heights don't have to be manually set once `HasUnevenRows` has been set to `true`, because the heights will be automatically calculated by Xamarin.Forms.

C#:

```
RowHeightDemoListView.HasUnevenRows = true;
```

XAML:

```
<ListView x:Name="RowHeightDemoListView" HasUnevenRows="true" />
```



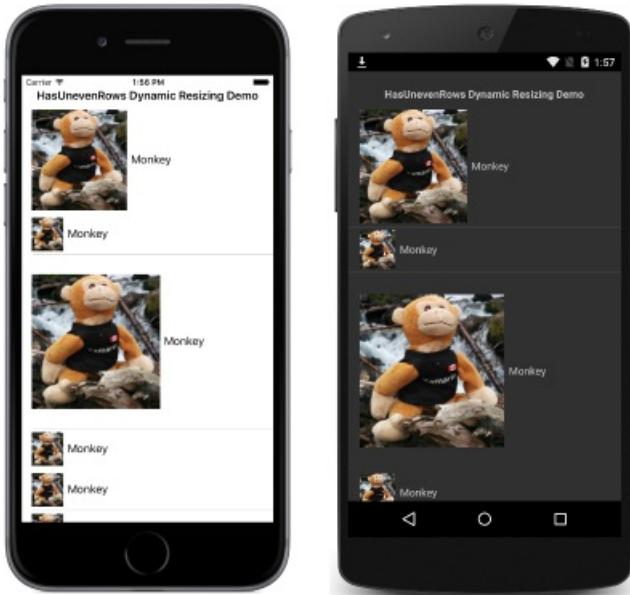
## Runtime Resizing of Rows

Individual `ListView` rows can be programmatically resized at runtime, provided that the `HasUnevenRows` property is set to `true`. The `Cell.ForceUpdateSize` method updates a cell's size, even when it isn't currently visible, as demonstrated in the following code example:

```
void OnImageTapped (object sender, EventArgs args)
{
    var image = sender as Image;
    var viewCell = image.Parent.Parent as ViewCell;

    if (image.HeightRequest < 250) {
        image.HeightRequest = image.Height + 100;
        viewCell.ForceUpdateSize ();
    }
}
```

The `OnImageTapped` event handler is executed in response to an `Image` in a cell being tapped, and increases the size of the `Image` displayed in the cell so that it's easily viewed.



Note that there is a strong possibility of performance degradation if this feature is overused.

## Related Links

- [Grouping \(sample\)](#)
- [Custom Renderer View \(sample\)](#)
- [Dynamic Resizing of Rows \(sample\)](#)
- [1.4 release notes](#)
- [1.3 release notes](#)

# ListView Interactivity

7/23/2018 • 4 minutes to read • [Edit Online](#)

ListView supports interacting with the data it presents through the following approaches:

- **Selection & Taps** – respond to taps and selections/deselections of items. Enable or disable row selection (enabled by default).
- **Context Actions** – Expose functionality per item, for example, swipe-to-delete.
- **Pull-to-Refresh** – Implement the pull-to-refresh idiom that users have come to expect from native experiences.

## Selection & Taps

The `ListView` selection mode is controlled by setting the `ListView.SelectionMode` property to a value of the `ListViewSelectionMode` enumeration:

- `Single` indicates that a single item can be selected, with the selected item being highlighted. This is the default value.
- `None` indicates that items cannot be selected.

When a user taps an item, two events are fired:

- `ItemSelected` fires when a new item is selected.
- `ItemTapped` fires when an item is tapped.

### NOTE

Tapping the same item twice will fire two `ItemTapped` events, but will only fire a single `ItemSelected` event.

When the `SelectionMode` property is set to `Single`, items in the `ListView` can be selected, the `ItemSelected` and `ItemTapped` events will be fired, and the `SelectedItem` property will be set to the value of the selected item.

When the `SelectionMode` property is set to `None`, items in the `ListView` cannot be selected, the `ItemSelected` event will not be fired, and the `SelectedItem` property will remain `null`. However, `ItemTapped` events will still be fired and the tapped item will be briefly highlighted during the tap.

When an item has been selected and the `SelectionMode` property is changed from `Single` to `None`, the `SelectedItem` property will be set to `null` and the `ItemSelected` event will be fired with a `null` item.

The following screenshots show a `ListView` with the default selection mode:



## Disabling Selection

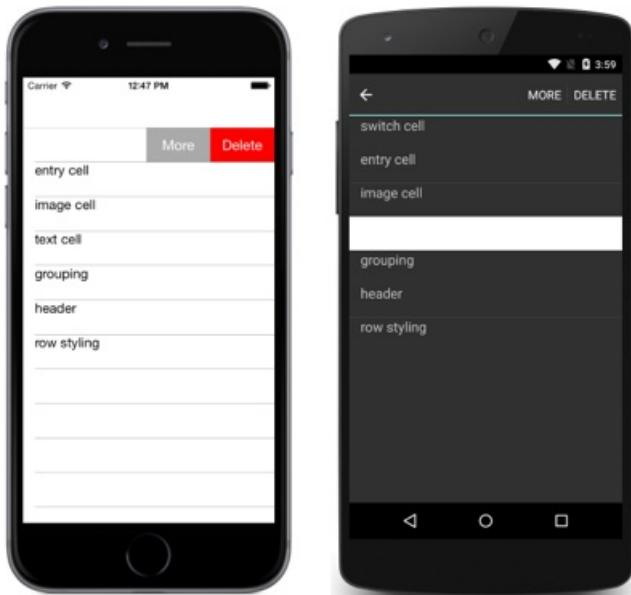
To disable `ListView` selection set the `SelectionMode` property to `None`:

```
<ListView ... SelectionMode="None" />
```

```
var listView = new ListView { ... SelectionMode = ListViewSelectionMode.None };
```

## Context Actions

Often, users will want to take action on an item in a `ListView`. For example, consider a list of emails in the Mail app. On iOS, you can swipe to delete a message:



Context actions can be implemented in C# and XAML. Below you'll find specific guides for both, but first let's take a look at some key implementation details for both.

Context Actions are created using `MenuItem`s. Tap events for `MenuItem`s are raised by the `MenuItem` itself, not the `ListView`. This is different from how tap events are handled for cells, where the `ListView` raises the event rather than the cell. Because the `ListView` is raising the event, its event handler is given key information, like which item was selected or tapped.

By default, a MenuItem has no way of knowing which cell it belongs to. `CommandParameter` is available on `MenuItem` to store objects, such as the object behind the MenuItem's ViewCell. `CommandParameter` can be set in both XAML and C#.

## C#

Context actions can be implemented in any `cell` subclass (as long as it isn't being used as a group header) by creating `MenuItem`s and adding them to the `ContextActions` collection for the cell. You have the following properties can be configured for the context action:

- **Text** – the string that appears in the menu item.
- **Clicked** – the event when the item is clicked.
- **IsDestructive** – (optional) when true the item is rendered differently on iOS.

Multiple context actions can be added to a cell, however only one should have `IsDestructive` set to `true`. The following code demonstrates how context actions would be added to a `ViewCell`:

```
var moreAction = new MenuItem { Text = "More" };
moreAction.SetBinding (MenuItem.CommandParameterProperty, new Binding ("."));
moreAction.Clicked += async (sender, e) => {
    var mi = ((MenuItem)sender);
    Debug.WriteLine("More Context Action clicked: " + mi.CommandParameter);
};

var deleteAction = new MenuItem { Text = "Delete", IsDestructive = true }; // red background
deleteAction.SetBinding (MenuItem.CommandParameterProperty, new Binding ("."));
deleteAction.Clicked += async (sender, e) => {
    var mi = ((MenuItem)sender);
    Debug.WriteLine("Delete Context Action clicked: " + mi.CommandParameter);
};
// add to the ViewCell's ContextActions property
ContextActions.Add (moreAction);
ContextActions.Add (deleteAction);
```

## XAML

`MenuItem`s can also be created in a XAML collection declaratively. The XAML below demonstrates a custom cell with two context actions implemented:

```
<ListView x:Name="ContextDemoList">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.ContextActions>
                    <MenuItem Clicked="OnMore" CommandParameter="{Binding .}"
                        Text="More" />
                    <MenuItem Clicked="OnDelete" CommandParameter="{Binding .}"
                        Text="Delete" IsDestructive="True" />
                </ViewCell.ContextActions>
                <StackLayout Padding="15,0">
                    <Label Text="{Binding title}" />
                </StackLayout>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

In the code-behind file, ensure the `Clicked` methods are implemented:

```
public void OnMore (object sender, EventArgs e) {
    var mi = ((MenuItem)sender);
    DisplayAlert("More Context Action", mi.CommandParameter + " more context action", "OK");
}

public void OnDelete (object sender, EventArgs e) {
    var mi = ((MenuItem)sender);
    DisplayAlert("Delete Context Action", mi.CommandParameter + " delete context action", "OK");
}
```

#### NOTE

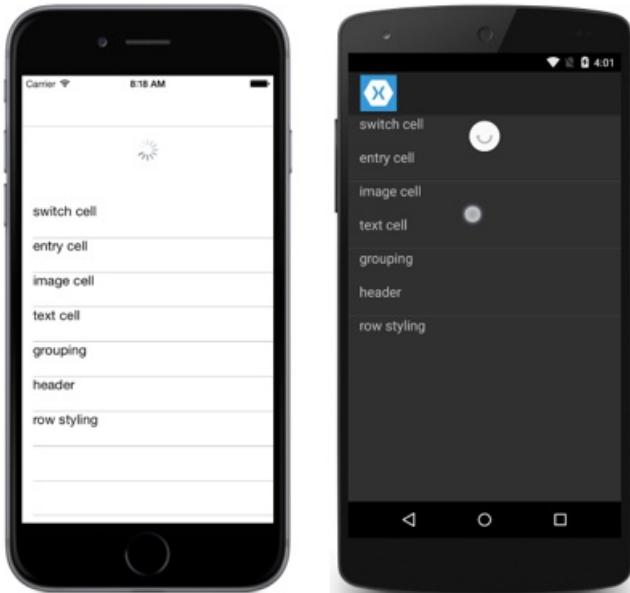
The `NavigationPageRenderer` for Android has an overridable `UpdateMenuItemIcon` method that can be used to load icons from a custom `Drawable`. This override makes it possible to use SVG images as icons on `MenuItem` instances on Android.

## Pull to Refresh

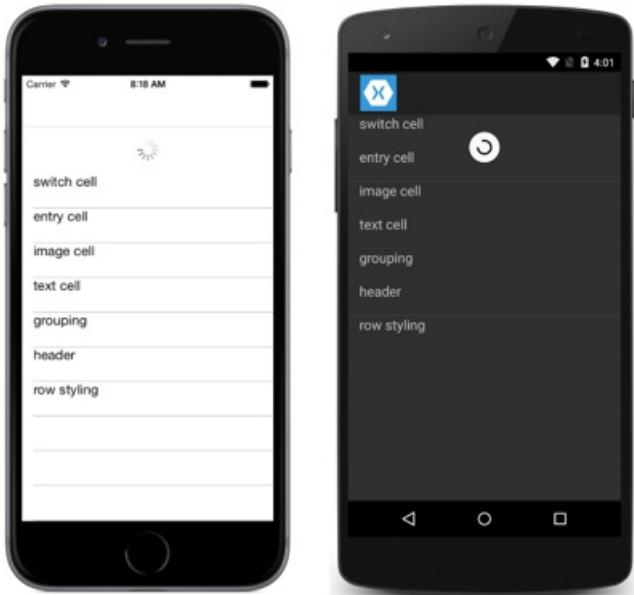
Users have come to expect that pulling down on a list of data will refresh that list. `ListView` supports this out-of-the-box. To enable pull-to-refresh functionality, set `IsPullToRefreshEnabled` to true:

```
listView.IsPullToRefreshEnabled = true;
```

Pull-to-Refresh as the user is pulling:



Pull-to-Refresh as the user has released the pull. This is what the user sees while you're updating list:



ListView exposes a few events that allow you to respond to pull-to-refresh events.

- The `RefreshCommand` will be invoked and the `Refreshing` event called. `IsRefreshing` will be set to `true`.
- You should perform whatever code is required to refresh the contents of the list view, either in the command or event.
- When refreshing is complete, call `EndRefresh` or set `IsRefreshing` to `false` to tell the list view that you're done.

The `CanExecute` property is respected, which gives you a way to control whether the pull-to-refresh command should be enabled.

## Related Links

- [ListView Interactivity \(sample\)](#)
- [1.4 release notes](#)
- [1.3 release notes](#)

# ListView Performance

7/12/2018 • 7 minutes to read • [Edit Online](#)

When writing mobile applications, performance matters. Users have come to expect smooth scrolling and fast load times. Failing to meet your users' expectations will cost you ratings in the application store, or in the case of a line-of-business application, cost your organization time and money.

Although `ListView` is a powerful view for displaying data, it has some limitations. Scrolling performance can suffer when using custom cells, especially when they contain deeply nested view hierarchies or use certain layouts that require a lot of measurement. Fortunately, there are techniques you can use to avoid poor performance.

## Caching Strategy

ListsViews are often used to display much more data than can fit onscreen. Consider a music app, for example. A library of songs may have thousands of entries. The simple approach, which would be to create a row for every song, would have poor performance. That approach wastes valuable memory and can slow scrolling to a crawl. Another approach is to create and destroy rows as data is scrolled into view. This requires constant instantiation and cleanup of view objects, which can be very slow.

To conserve memory, the native `ListView` equivalents for each platform have built-in features for re-using rows. Only the cells visible on screen are loaded in memory and the **content** is loaded into existing cells. This prevents the application from needing to instantiate thousands of objects, saving time and memory.

Xamarin.Forms permits `ListView` cell re-use through the `ListViewCachingStrategy` enumeration, which has the following values:

```
public enum ListViewCachingStrategy
{
    RetainElement, // the default value
    RecycleElement,
    RecycleElementAndDataTemplate
}
```

### NOTE

The Universal Windows Platform (UWP) ignores the `RetainElement` caching strategy, because it always uses caching to improve performance. Therefore, by default it behaves as if the `RecycleElement` caching strategy is applied.

### RetainElement

The `RetainElement` caching strategy specifies that the `ListView` will generate a cell for each item in the list, and is the default `ListView` behavior. It should generally be used in the following circumstances:

- When each cell has a large number of bindings (20-30+).
- When the cell template changes frequently.
- When testing reveals that the `RecycleElement` caching strategy results in a reduced execution speed.

It's important to recognize the consequences of the `RetainElement` caching strategy when working with custom cells. Any cell initialization code will need to run for each cell creation, which may be multiple times per second. In this circumstance, layout techniques that were fine on a page, like using multiple nested `StackLayout` instances, become performance bottlenecks when they are setup and destroyed in real time as the user scrolls.

## RecycleElement

The `RecycleElement` caching strategy specifies that the `ListView` will attempt to minimize its memory footprint and execution speed by recycling list cells. This mode does not always offer a performance improvement, and testing should be performed to determine any improvements. However, it is generally the preferred choice, and should be used in the following circumstances:

- When each cell has a small to moderate number of bindings.
- When each cell's `BindingContext` defines all of the cell data.
- When each cell is largely similar, with the cell template unchanging.

During virtualization the cell will have its binding context updated, and so if an application uses this mode it must ensure that binding context updates are handled appropriately. All data about the cell must come from the binding context or consistency errors may occur. This can be accomplished by using data binding to display cell data.

Alternatively, cell data should be set in the `OnBindingContextChanged` override, rather than in the custom cell's constructor, as demonstrated in the following code example:

```
public class CustomCell : ViewCell
{
    Image image = null;

    public CustomCell ()
    {
        image = new Image();
        View = image;
    }

    protected override void OnBindingContextChanged ()
    {
        base.OnBindingContextChanged ();

        var item = BindingContext as ImageItem;
        if (item != null) {
            image.Source = item.ImageUrl;
        }
    }
}
```

For more information, see [Binding Context Changes](#).

On iOS and Android, if cells use custom renderers, they must ensure that property change notification is correctly implemented. When cells are reused their property values will change when the binding context is updated to that of an available cell, with `PropertyChanged` events being raised. For more information, see [Customizing a ViewCell](#).

### RecycleElement with a DataTemplateSelector

When a `ListView` uses a `DataTemplateSelector` to select a `DataTemplate`, the `RecycleElement` caching strategy does not cache `DataTemplate`s. Instead, a `DataTemplate` is selected for each item of data in the list.

#### NOTE

The `RecycleElement` caching strategy has a pre-requisite, introduced in Xamarin.Forms 2.4, that when a `DataTemplateSelector` is asked to select a `DataTemplate` that each `DataTemplate` must return the same `ViewCell` type. For example, given a `ListView` with a `DataTemplateSelector` that can return either `MyDataTemplateA` (where `MyDataTemplateA` returns a `ViewCell` of type `MyViewCellA`), or `MyDataTemplateB` (where `MyDataTemplateB` returns a `ViewCell` of type `MyViewCellB`), when `MyDataTemplateA` is returned it must return `MyViewCellA` or an exception will be thrown.

## RecycleElementAndDataTemplate

The `RecycleElementAndDataTemplate` caching strategy builds on the `RecycleElement` caching strategy by additionally ensuring that when a `ListView` uses a `DataTemplateSelector` to select a `DataTemplate`, `DataTemplate`s are cached by the type of item in the list. Therefore, `DataTemplate`s are selected once per item type, instead of once per item instance.

#### NOTE

The `RecycleElementAndDataTemplate` caching strategy has a pre-requisite that the `DataTemplate`s returned by the `DataTemplateSelector` must use the `DataTemplate` constructor that takes a `Type`.

## Setting the Caching Strategy

The `ListViewCachingStrategy` enumeration value is specified with a `ListView` constructor overload, as shown in the following code example:

```
var listView = new ListView(ListViewCachingStrategy.RecycleElement);
```

In XAML, set the `CachingStrategy` attribute as shown in the code below:

```
<ListView CachingStrategy="RecycleElement">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                ...
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

This has the same effect as setting the caching strategy argument in the constructor in C#; note that there is no `CachingStrategy` property on `ListView`.

### Setting the Caching Strategy in a Subclassed ListView

Setting the `CachingStrategy` attribute from XAML on a subclassed `ListView` will not produce the desired behavior, because there is no `CachingStrategy` property on `ListView`. In addition, if `XAMLC` is enabled, the following error message will be produced: **No property, bindable property, or event found for 'CachingStrategy'**

The solution to this issue is to specify a constructor on the subclassed `ListView` that accepts a `ListViewCachingStrategy` parameter and passes it into the base class:

```
public class CustomListView : ListView
{
    public CustomListView (ListViewCachingStrategy strategy) : base (strategy)
    {
    }
    ...
}
```

Then the `ListViewCachingStrategy` enumeration value can be specified from XAML by using the `x:Arguments` syntax:

```
<local:CustomListView>
<x:Arguments>
    <ListViewCachingStrategy>RecycleElement</ListViewCachingStrategy>
</x:Arguments>
</local:CustomListView>
```

## Improving ListView Performance

There are many techniques for improving the performance of a `ListView`:

- Bind the `ItemsSource` property to an `IList<T>` collection instead of an `IEnumerable<T>` collection, because `IEnumerable<T>` collections don't support random access.
- Use the built-in cells (like `TextCell` / `SwitchCell`) instead of `ViewCell` whenever you can.
- Use fewer elements. For example consider using a single `FormattedString` label instead of multiple labels.
- Replace the `ListView` with a `TableView` when displaying non-homogenous data – that is, data of different types.
- Limit the use of the `Cell.ForceUpdateSize` method. If overused, it will degrade performance.
- On Android, avoid setting a `ListView`'s row separator visibility or color after it has been instantiated, as it results in a large performance penalty.
- Avoid changing the cell layout based on the `BindingContext`. This incurs large layout and initialization costs.
- Avoid deeply nested layout hierarchies. Use `AbsoluteLayout` or `Grid` to help reduce nesting.
- Avoid specific `LayoutOptions` other than `Fill` (Fill is the cheapest to compute).
- Avoid placing a `ListView` inside a `ScrollView` for the following reasons:
  - The `ListView` implements its own scrolling.
  - The `ListView` will not receive any gestures, as they will be handled by the parent `ScrollView`.
  - The `ListView` can present a customized header and footer that scrolls with the elements of the list, potentially offering the functionality that the `ScrollView` was used for. For more information see [Headers and Footers](#).
- Consider a custom renderer if you need a very specific, complex design presented in your cells.

`AbsoluteLayout` has the potential to perform layouts without a single measure call. This makes it very powerful for performance. If `AbsoluteLayout` cannot be used, consider `RelativeLayout`. If using `RelativeLayout`, passing Constraints directly will be considerably faster than using the expression API. That is because the expression API uses JIT, and on iOS the tree has to be interpreted, which is slower. The expression API is suitable for page layouts where it only required on initial layout and rotation, but in `ListView`, where it's run constantly during scrolling, it hurts performance.

Building a custom renderer for a `ListView` or its cells is one approach to reducing the effect of layout calculations on scrolling performance. For more information, see [Customizing a ListView](#) and [Customizing a ViewCell](#).

## Related Links

- [Custom Renderer View \(sample\)](#)
- [Custom Renderer ViewCell \(sample\)](#)
- [ListViewCachingStrategy](#)

# Xamarin.Forms Map

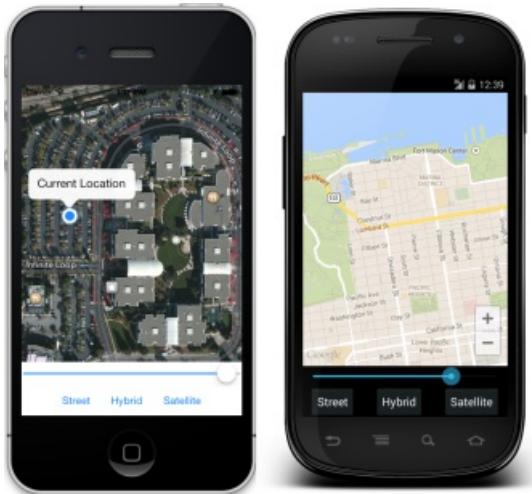
11/11/2018 • 6 minutes to read • [Edit Online](#)

*Xamarin.Forms uses the native map APIs on each platform.*

Xamarin.Forms.Maps uses the native map APIs on each platform. This provides a fast, familiar maps experience for users, but means that some configuration steps are needed to adhere to each platforms specific API requirements. Once configured, the `Map` control works just like any other Xamarin.Forms element in common code.

- [Maps Initialization](#) - Using `Map` requires additional initialization code at startup.
- [Platform Configuration](#) - Each platform requires some configuration for maps to work.
- [Using Maps in C#](#) - Displaying maps and pins using C#.
- [Using Maps in XAML](#) - Displaying a map with XAML.

The map control has been used in the [MapsSample](#) sample, which is shown below.



Map functionality can be further enhanced by creating a [map custom renderer](#).

## Maps Initialization

When adding maps to a Xamarin.Forms application, **Xamarin.Forms.Maps** is a separate NuGet package that you should add to every project in the solution. On Android, this also has a dependency on GooglePlayServices (another NuGet) which is downloaded automatically when you add Xamarin.Forms.Maps.

After installing the NuGet package, some initialization code is required in each application project, *after* the `Xamarin.Forms.Forms.Init` method call. For iOS use the following code:

```
Xamarin.FormsMaps.Init();
```

On Android you must pass the same parameters as `Forms.Init`:

```
Xamarin.FormsMaps.Init(this, bundle);
```

For the Universal Windows Platform (UWP) use the following code:

```
Xamarin.FormsMaps.Init("INSERT_AUTHENTICATION_TOKEN_HERE");
```

Add this call in the following files for each platform:

- **iOS** - AppDelegate.cs file, in the `FinishedLaunching` method.
- **Android** - MainActivity.cs file, in the `OnCreate` method.
- **UWP** - MainPage.xaml.cs file, in the `MainPage` constructor.

Once the NuGet package has been added and the initialization method called inside each application, `Xamarin.Forms.Maps` APIs can be used in the common .NET Standard library project or Shared Project code.

## Platform Configuration

Additional configuration steps are required on some platforms before the map will display.

### iOS

To access location services on iOS, you must set the following keys in **Info.plist**:

- iOS 11
  - `NSLocationWhenInUseUsageDescription` – for using location services when the app is in use
  - `NSLocationAlwaysAndWhenInUseUsageDescription` – for using location services at all times
- iOS 10 and earlier
  - `NSLocationWhenInUseUsageDescription` – for using location services when the app is in use
  - `NSLocationAlwaysUsageDescription` – for using location services at all times

To support iOS 11 and earlier, you can include all three keys: `NSLocationWhenInUseUsageDescription`, `NSLocationAlwaysAndWhenInUseUsageDescription`, and `NSLocationAlwaysUsageDescription`.

The XML representation for these keys in **Info.plist** is shown below. You should update the `string` values to reflect how your application is using the location information:

```
<key>NSLocationAlwaysUsageDescription</key>
<string>Can we use your location at all times?</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>Can we use your location when your app is being used?</string>
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>Can we use your location at all times?</string>
```

The **Info.plist** entries can also be added in **Source** view while editing the **Info.plist** file:

<code>NSLocationWhenInUseUsageDescription</code>	<code>String</code>	We are using your location
<code>NSLocationAlwaysUsageDescription</code>	<code>String</code>	Can we use your location

### Android

To use the [Google Maps API v2](#) on Android you must generate an API key and add it to your Android project. Follow the instructions in the Xamarin doc on [obtaining a Google Maps API v2 key](#). After following those instructions, paste the API key in the **Properties/AndroidManifest.xml** file (view source and find/update the following element):

```
<application ...>
    <meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="YOUR_API_KEY" />
</application>
```

Without a valid API key the maps control will display as a grey box on Android.

#### NOTE

Note that, in order for your APK to access Google Maps, you must include SHA-1 fingerprints and package names for every keystore (debug and release) that you use to sign your APK. For example, if you use one computer for debug and another computer for generating the release APK, you should include the SHA-1 certificate fingerprint from the debug keystore of the first computer and the SHA-1 certificate fingerprint from the release keystore of the second computer. Also remember to edit the key credentials if the app's **Package Name** changes. See [obtaining a Google Maps API v2 key](#).

You'll also need to enable appropriate permissions by right-clicking on the Android project and selecting **Options > Build > Android Application** and ticking the following:

- `AccessCoarseLocation`
- `AccessFineLocation`
- `AccessLocationExtraCommands`
- `AccessMockLocation`
- `AccessNetworkState`
- `AccessWifiState`
- `Internet`

Some of these are shown in the screenshot below:



The last two are required because applications require a network connection to download map data. Read about Android [permissions](#) to learn more.

#### Universal Windows Platform

To use maps on the Universal Windows Platform you must generate an authorization token. For more information, see [Request a maps authentication key](#) on MSDN.

The authentication token should then be specified in the `FormsMaps.Init("AUTHORIZATION_TOKEN")` method call, to authenticate the app with Bing Maps.

## Using Maps

See the `MapPage.cs` in the MobileCRM sample for an example of how the map control can be used in code. A simple `MapPage` class might look like this - notice that a new `MapSpan` is created to position the map's view:

```

public class MapPage : ContentPage {
    public MapPage() {
        var map = new Map(
            MapSpan.FromCenterAndRadius(
                new Position(37, -122), Distance.FromMiles(0.3))) {
            IsShowingUser = true,
            HeightRequest = 100,
            WidthRequest = 960,
            VerticalOptions = LayoutOptions.FillAndExpand
        };
        var stack = new StackLayout { Spacing = 0 };
        stack.Children.Add(map);
        Content = stack;
    }
}

```

## Map Type

The map content can also be changed by setting the `MapType` property, to show a regular street map (the default), satellite imagery or a combination of both.

```
map.MapType == MapType.Street;
```

Valid `MapType` values are:

- Hybrid
- Satellite
- Street (the default)

## Map Region and MapSpan

As shown in the code snippet above, supplying a `MapSpan` instance to a map constructor sets the initial view (center point and zoom level) of the map when it is loaded. The `MoveToRegion` method on the map class can then be used to change the position or zoom level of the map. There are two ways to create a new `MapSpan` instance:

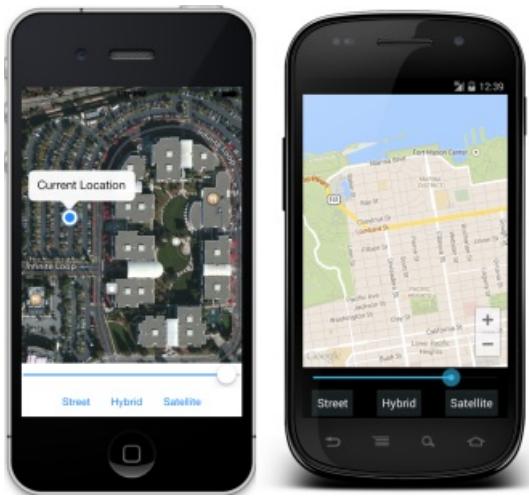
- **MapSpan.FromCenterAndRadius()** - static method to create a span from a `Position` and specifying a `Distance`.
- **new MapSpan ()** - constructor that uses a `Position` and the degrees of latitude and longitude to display.

To change the zoom level of the map without altering the location, create a new `MapSpan` using the current location from the `VisibleRegion.Center` property of the map control. A `slider` could be used to control map zoom like this (however zooming directly in the map control cannot currently update the value of the slider):

```

var slider = new Slider (1, 18, 1);
slider.ValueChanged += (sender, e) => {
    var zoomLevel = e.NewValue; // between 1 and 18
    var latlongdegrees = 360 / (Math.Pow(2, zoomLevel));
    map.MoveToRegion(new MapSpan (map.VisibleRegion.Center, latlongdegrees, latlongdegrees));
};

```



## Map Pins

Locations can be marked on the map with `Pin` objects.

```
var position = new Position(37,-122); // Latitude, Longitude
var pin = new Pin {
    Type = PinType.Place,
    Position = position,
    Label = "custom pin",
    Address = "custom detail info"
};
map.Pins.Add(pin);
```

`PinType` can be set to one of the following values, which may affect the way the pin is rendered (depending on the platform):

- Generic
- Place
- SavedPin
- SearchResult

## Using Xaml

Maps can also be positioned in Xaml layouts as shown in this snippet.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
    x:Class="MapDemo.MapPage">
    <StackLayout VerticalOptions="StartAndExpand" Padding="30">
        <maps:Map WidthRequest="320" HeightRequest="200"
            x:Name="MyMap"
            IsShowingUser="true"
            MapType="Hybrid"
        />
    </StackLayout>
</ContentPage>
```

The `MapRegion` and `Pins` can be set in code using the `MyMap` reference (or whatever the map is named). Note that an additional `xmllns` namespace definition is required to reference the `Xamarin.Forms.Maps` controls.

```
MyMap.MoveToRegion(  
    MapSpan.FromCenterAndRadius(  
        new Position(37,-122), Distance.FromMiles(1)));
```

## Summary

The Xamarin.Forms.Maps is a separate NuGet that must be added to each project in a Xamarin.Forms solution. Additional initialization code is required, as well as some configuration steps for iOS, Android, and UWP.

Once configured the Maps API can be used to render maps with pin markers in just a few lines of code. Maps can be further enhanced with a [custom renderer](#).

## Related Links

- [MapsSample](#)
- [Map Custom Renderer](#)
- [Xamarin.Forms Samples](#)

# Xamarin.Forms Picker

7/12/2018 • 2 minutes to read • [Edit Online](#)

The Picker view is a control for selecting a text item from a list of data.

The Xamarin.Forms `Picker` displays a short list of items, from which the user can select an item. `Picker` defines eight properties:

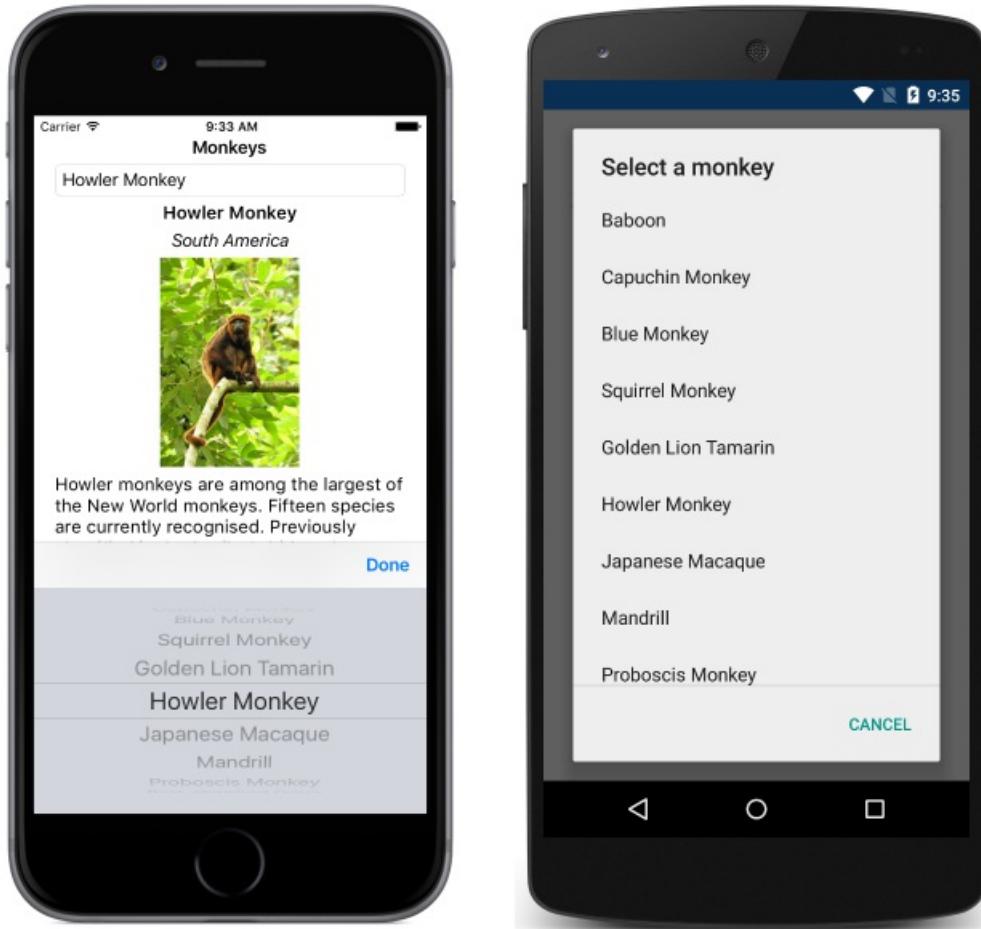
- `Title` of type `string`, which defaults to `null`.
- `ItemsSource` of type `IList`, the source list of items to display, which defaults to `null`.
- `SelectedIndex` of type `int`, the index of the selected item, which defaults to -1.
- `SelectedItem` of type `object`, the selected item, which defaults to `null`.
- `TextColor` of type `Color`, the color used to display the text, which defaults to `Color.Default`.
- `FontAttributes` of type `FontAttributes`, which defaults to `FontAttributes.None`.
- `FontFamily` of type `string`, which defaults to `null`.
- `FontSize` of type `double`, which defaults to -1.0.

All eight properties are backed by `BindableProperty` objects, which means that they can be styled, and the properties can be targets of data bindings. The `SelectedIndex` and `SelectedItem` properties have a default binding mode of `BindingMode.TwoWay`, which means that they can be targets of data bindings in an application that uses the Model-View-ViewModel (MVVM) architecture. For information about setting font properties, see [Fonts](#).

A `Picker` doesn't show any data when it's first displayed. Instead, the value of its `Title` property is shown as a placeholder on the iOS and Android platforms:



When the `Picker` gains focus, its data is displayed and the user can select an item:



The `Picker` fires a `SelectedIndexChanged` event when the user selects an item. Following selection, the selected item is displayed by the `Picker`:



There are two techniques for populating a `Picker` with data:

- Setting the `ItemsSource` property to the data to be displayed. This is the recommended technique, which was introduced in Xamarin.Forms 2.3.4. For more information, see [Setting a Picker's ItemsSource Property](#).
- Adding the data to be displayed to the `Items` collection. This technique was the original process for populating a `Picker` with data. For more information, see [Adding Data to a Picker's Items Collection](#).

## Related Links

- [Picker](#)

# Setting a Picker's ItemsSource Property

7/12/2018 • 4 minutes to read • [Edit Online](#)

The Picker view is a control for selecting a text item from a list of data. This article explains how to populate a Picker with data by setting the `ItemsSource` property, and how to respond to item selection by the user.

Xamarin.Forms 2.3.4 has enhanced the `Picker` view by adding the ability to populate it with data by setting its `ItemsSource` property, and to retrieve the selected item from the `SelectedItem` property. In addition, the color of the text for the selected item can be changed by setting the `TextColor` property to a `Color`.

## Populating a Picker with Data

A `Picker` can be populated with data by setting its `ItemsSource` property to an `IList` collection. Each item in the collection must be of, or derived from, type `object`. Items can be added in XAML by initializing the `ItemsSource` property from an array of items:

```
<Picker x:Name="picker" Title="Select a monkey">
  <Picker.ItemsSource>
    <x:Array Type="{x:Type x:String}">
      <x:String>Baboon</x:String>
      <x:String>Capuchin Monkey</x:String>
      <x:String>Blue Monkey</x:String>
      <x:String>Squirrel Monkey</x:String>
      <x:String>Golden Lion Tamarin</x:String>
      <x:String>Howler Monkey</x:String>
      <x:String>Japanese Macaque</x:String>
    </x:Array>
  </Picker.ItemsSource>
</Picker>
```

### NOTE

Note that the `x:Array` element requires a `Type` attribute indicating the type of the items in the array.

The equivalent C# code is shown below:

```
var monkeyList = new List<string>();
monkeyList.Add("Baboon");
monkeyList.Add("Capuchin Monkey");
monkeyList.Add("Blue Monkey");
monkeyList.Add("Squirrel Monkey");
monkeyList.Add("Golden Lion Tamarin");
monkeyList.Add("Howler Monkey");
monkeyList.Add("Japanese Macaque");

var picker = new Picker { Title = "Select a monkey" };
picker.ItemsSource = monkeyList;
```

## Responding to Item Selection

A `Picker` supports selection of one item at a time. When a user selects an item, the `SelectedIndexChanged` event fires, the `SelectedIndex` property is updated to an integer representing the index of the selected item in the list,

and the `SelectedItem` property is updated to the `object` representing the selected item. The `SelectedIndex` property is a zero-based number indicating the item the user selected. If no item is selected, which is the case when the `Picker` is first created and initialized, `SelectedIndex` will be -1.

#### NOTE

Item selection behavior in a `Picker` can be customized on iOS with a platform-specific. For more information, see [Controlling Picker Item Selection](#).

The following code example shows how to retrieve the `SelectedItem` property value from the `Picker` in XAML:

```
<Label Text="{Binding Source={x:Reference picker}, Path=SelectedItem}" />
```

The equivalent C# code is shown below:

```
var monkeyNameLabel = new Label();
monkeyNameLabel.SetBinding(Label.TextProperty, new Binding("SelectedItem", source: picker));
```

In addition, an event handler can be executed when the `SelectedIndexChanged` event fires:

```
void OnPickerSelectedIndexChanged(object sender, EventArgs e)
{
    var picker = (Picker)sender;
    int selectedIndex = picker.SelectedIndex;

    if (selectedIndex != -1)
    {
        monkeyNameLabel.Text = (string)picker.ItemsSource[selectedIndex];
    }
}
```

This method obtains the `SelectedIndex` property value, and uses the value to retrieve the selected item from the `ItemsSource` collection. This is functionally equivalent to retrieving the selected item from the `SelectedItem` property. Note that each item in the `ItemsSource` collection is of type `object`, and so must be cast to a `string` for display.

#### NOTE

A `Picker` can be initialized to display a specific item by setting the `SelectedIndex` or `SelectedItem` properties. However, these properties must be set after initializing the `ItemsSource` collection.

## Populating a Picker with Data Using Data Binding

A `Picker` can also be populated with data by using data binding to bind its `ItemsSource` property to an `IList` collection. In XAML this is achieved with the `Binding` markup extension:

```
<Picker Title="Select a monkey" ItemsSource="{Binding Monkeys}" ItemDisplayBinding="{Binding Name}" />
```

The equivalent C# code is shown below:

```

var picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.ItemDisplayBinding = new Binding("Name");

```

The `ItemsSource` property data binds to the `Monkeys` property of the connected view model, which returns an `IList<Monkey>` collection. The following code example shows the `Monkey` class, which contains four properties:

```

public class Monkey
{
    public string Name { get; set; }
    public string Location { get; set; }
    public string Details { get; set; }
    public string ImageUrl { get; set; }
}

```

When binding to a list of objects, the `Picker` must be told which property to display from each object. This is achieved by setting the `ItemDisplayBinding` property to the required property from each object. In the code examples above, the `Picker` is set to display each `Monkey.Name` property value.

## Responding to Item Selection

Data binding can be used to set an object to the `SelectedItem` property value when it changes:

```

<Picker Title="Select a monkey"
        ItemsSource="{Binding Monkeys}"
        ItemDisplayBinding="{Binding Name}"
        SelectedItem="{Binding SelectedMonkey}" />
<Label Text="{Binding SelectedMonkey.Name}" ... />
<Label Text="{Binding SelectedMonkey.Location}" ... />
<Image Source="{Binding SelectedMonkey.ImageUrl}" ... />
<Label Text="{Binding SelectedMonkey.Details}" ... />

```

The equivalent C# code is shown below:

```

var picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.SetBinding(Picker.SelectedItemProperty, "SelectedMonkey");
picker.ItemDisplayBinding = new Binding("Name");

var nameLabel = new Label { ... };
nameLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Name");

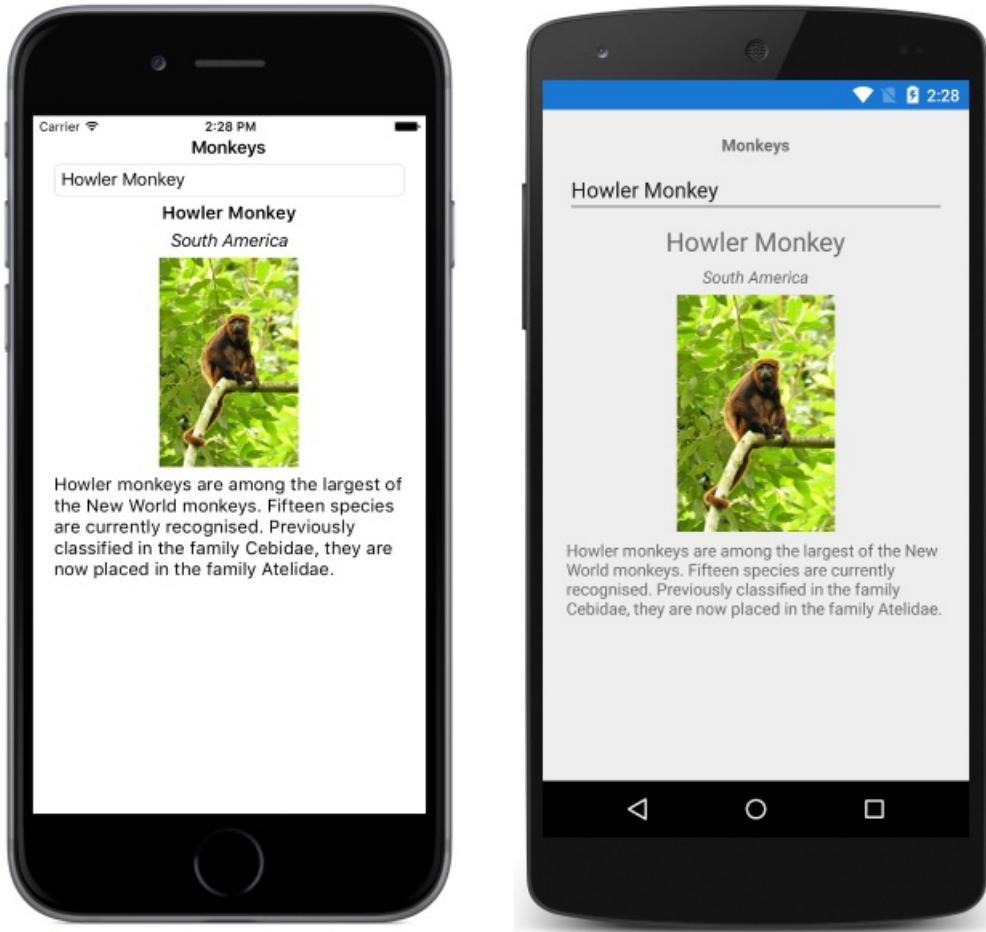
var locationLabel = new Label { ... };
locationLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Location");

var image = new Image { ... };
image.SetBinding(Image.SourceProperty, "SelectedMonkey.ImageUrl");

var detailsLabel = new Label();
detailsLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Details");

```

The `SelectedItem` property data binds to the `SelectedMonkey` property of the connected view model, which is of type `Monkey`. Therefore, when the user selects an item in the `Picker`, the `SelectedMonkey` property will be set to the selected `Monkey` object. The `SelectedMonkey` object data is displayed in the user interface by `Label` and `Image` views:



#### NOTE

Note that the `SelectedItem` and `SelectedIndex` properties both support two-way bindings by default.

## Summary

The `Picker` view is a control for selecting a text item from a list of data. This article explained how to populate a `Picker` with data by setting the `ItemsSource` property, and how to respond to item selection by the user. This approach, which was introduced in Xamarin.Forms 2.3.4, is the recommended approach for interacting with a `Picker`.

## Related Links

- [Picker Demo \(sample\)](#)
- [Monkey App \(sample\)](#)
- [Bindable Picker \(sample\)](#)
- [Picker](#)

# Adding Data to a Picker's Items Collection

7/12/2018 • 2 minutes to read • [Edit Online](#)

The `Picker` view is a control for selecting a text item from a list of data. This article explains how to populate a `Picker` with data by adding it to the `Items` collection, and how to respond to item selection by the user.

## Populating a Picker with Data

Prior to Xamarin.Forms 2.3.4, the process for populating a `Picker` with data was to add the data to be displayed to the read-only `Items` collection, which is of type `IList<string>`. Each item in the collection must be of type `string`. Items can be added in XAML by initializing the `Items` property with a list of `x:String` items:

```
<Picker Title="Select a monkey">
  <Picker.Items>
    <x:String>Baboon</x:String>
    <x:String>Capuchin Monkey</x:String>
    <x:String>Blue Monkey</x:String>
    <x:String>Squirrel Monkey</x:String>
    <x:String>Golden Lion Tamarin</x:String>
    <x:String>Howler Monkey</x:String>
    <x:String>Japanese Macaque</x:String>
  </Picker.Items>
</Picker>
```

The equivalent C# code is shown below:

```
var picker = new Picker { Title = "Select a monkey" };
picker.Items.Add("Baboon");
picker.Items.Add("Capuchin Monkey");
picker.Items.Add("Blue Monkey");
picker.Items.Add("Squirrel Monkey");
picker.Items.Add("Golden Lion Tamarin");
picker.Items.Add("Howler Monkey");
picker.Items.Add("Japanese Macaque");
```

In addition to adding data using the `Items.Add` method, data can also be inserted into the collection by using the `Items.Insert` method.

## Responding to Item Selection

A `Picker` supports selection of one item at a time. When a user selects an item, the `SelectedIndexChanged` event fires, and the `SelectedIndex` property is updated to an integer representing the index of the selected item in the list. The `SelectedIndex` property is a zero-based number indicating the item that the user selected. If no item is selected, which is the case when the `Picker` is first created and initialized, `SelectedIndex` will be -1.

### NOTE

Item selection behavior in a `Picker` can be customized on iOS with a platform-specific. For more information, see [Controlling Picker Item Selection](#).

The following code example shows the `OnPickerSelectedIndexChanged` event handler method, which is executed

when the `SelectedIndexChanged` event fires:

```
void OnPickerSelectedIndexChanged(object sender, EventArgs e)
{
    var picker = (Picker)sender;
    int selectedIndex = picker.SelectedIndex;

    if (selectedIndex != -1)
    {
        monkeyNameLabel.Text = picker.Items[selectedIndex];
    }
}
```

This method obtains the `SelectedIndex` property value, and uses the value to retrieve the selected item from the `Items` collection. Because each item in the `Items` collection is a `string`, they can be displayed by a `Label` without requiring a cast.

#### NOTE

A `Picker` can be initialized to display a specific item by setting the `SelectedIndex` property. However, the `SelectedIndex` property must be set after initializing the `Items` collection.

## Summary

The `Picker` view is a control for selecting a text item from a list of data. This article explained how to populate a `Picker` with data by adding it to the `Items` collection, and how to respond to item selection by the user. This was the process for using a `Picker` prior to Xamarin.Forms 2.3.4.

## Related Links

- [Picker Demo \(sample\)](#)
- [Picker](#)

# Xamarin.Forms Slider

11/20/2018 • 12 minutes to read • [Edit Online](#)

Use a `Slider` for selecting from a range of continuous values.

The Xamarin.Forms `Slider` is a horizontal bar that can be manipulated by the user to select a `double` value from a continuous range.

The `Slider` defines three properties of type `double`:

- `Minimum` is the minimum of the range, with a default value of 0.
- `Maximum` is the maximum of the range, with a default value of 1.
- `Value` is the slider's value, which can range between `Minimum` and `Maximum` and has a default value of 0.

All three properties are backed by `BindableProperty` objects. The `Value` property has a default binding mode of `BindingMode.TwoWay`, which means that it's suitable as a binding source in an application that uses the [Model-View-ViewModel \(MVVM\)](#) architecture.

## WARNING

Internally, the `Slider` ensures that `Minimum` is less than `Maximum`. If `Minimum` or `Maximum` are ever set so that `Minimum` is not less than `Maximum`, an exception is raised. See the [Precautions](#) section below for more information on setting the `Minimum` and `Maximum` properties.

The `Slider` coerces the `Value` property so that it is between `Minimum` and `Maximum`, inclusive. If the `Minimum` property is set to a value greater than the `Value` property, the `Slider` sets the `Value` property to `Minimum`. Similarly, if `Maximum` is set to a value less than `Value`, then `Slider` sets the `Value` property to `Maximum`.

The `Slider` defines a `ValueChanged` event that is fired when the `Value` changes, either through user manipulation of the `Slider` or when the program sets the `Value` property directly. A `valueChanged` event is also fired when the `Value` property is coerced as described in the previous paragraph.

The `ValueChangedEventArgs` object that accompanies the `ValueChanged` event has two properties, both of type `double`: `OldValue` and `NewValue`. At the time the event is fired, the value of `NewValue` is the same as the `Value` property of the `Slider` object.

## WARNING

Do not use unconstrained horizontal layout options of `Center`, `Start`, or `End` with `Slider`. On both Android and the UWP, the `Slider` collapses to a bar of zero length, and on iOS, the bar is very short. Keep the default `HorizontalOptions` setting of `Fill`, and don't use a width of `Auto` when putting `Slider` in a `Grid` layout.

The `Slider` also defines several properties that affect its appearance:

- `MinimumTrackColor` is the bar color on the left side of the thumb.
- `MaximumTrackColor` is the bar color on the right side of the thumb.
- `ThumbColor` is the thumb color.
- `ThumbImage` is the image to use for the thumb, of type `FileImageSource`.

#### NOTE

The `ThumbColor` and `ThumbImage` properties are mutually exclusive. If both properties are set, the `ThumbImage` property will take precedence.

## Basic Slider code and markup

The [SliderDemos](#) sample begins with three pages that are functionally identical, but are implemented in different ways. The first page uses only C# code, the second uses XAML with an event handler in code, and the third is able to avoid the event handler by using data binding in the XAML file.

### Creating a Slider in code

The **Basic Slider Code** page in the [SliderDemos](#) sample shows how to create a `Slider` and two `Label` objects in code:

```
public class BasicSliderCodePage : ContentPage
{
    public BasicSliderCodePage()
    {
        Label rotationLabel = new Label
        {
            Text = "ROTATING TEXT",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Label displayLabel = new Label
        {
            Text = "(uninitialized)",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

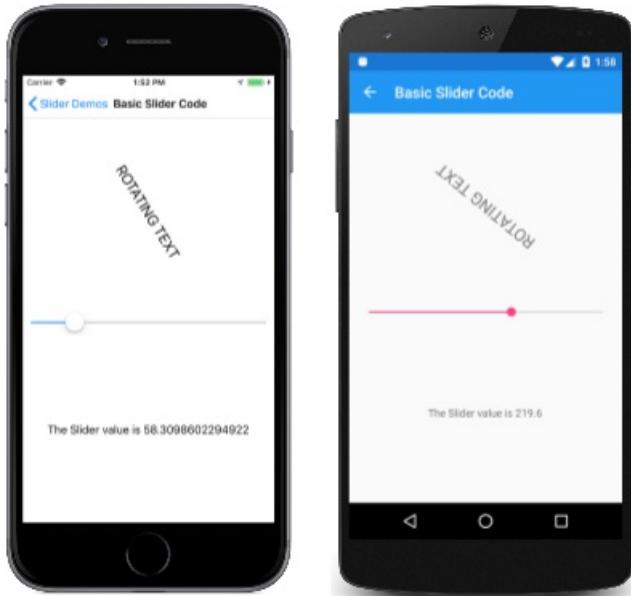
        Slider slider = new Slider
        {
            Maximum = 360
        };
        slider.ValueChanged += (sender, args) =>
        {
            rotationLabel.Rotation = slider.Value;
            displayLabel.Text = String.Format("The Slider value is {0}", args.NewValue);
        };

        Title = "Basic Slider Code";
        Padding = new Thickness(10, 0);
        Content = new StackLayout
        {
            Children =
            {
                rotationLabel,
                slider,
                displayLabel
            }
        };
    }
}
```

The `Slider` is initialized to have a `Maximum` property of 360. The `ValueChanged` handler of the `Slider` uses the `Value` property of the `slider` object to set the `Rotation` property of the first `Label` and uses the `String.Format` method with the `NewValue` property of the event arguments to set the `Text` property of the second `Label`. These

two approaches to obtain the current value of the `Slider` are interchangeable.

Here's the program running on iOS, Android, and Universal Windows Platform (UWP) devices:



The second `Label` displays the text "(uninitialized)" until the `Slider` is manipulated, which causes the first `ValueChanged` event to be fired. Notice that the number of decimal places that are displayed is different for each platform. These differences are related to the platform implementations of the `Slider` and are discussed later in this article in the section [Platform implementation differences](#).

### Creating a Slider in XAML

The **Basic Slider XAML** page is functionally the same as **Basic Slider Code** but implemented mostly in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SliderDemos.BasicSliderXamlPage"
    Title="Basic Slider XAML"
    Padding="10, 0">
    <StackLayout>
        <Label x:Name="rotatingLabel"
            Text="ROTATING TEXT"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Slider Maximum="360"
            ValueChanged="OnSliderValueChanged" />

        <Label x:Name="displayLabel"
            Text="(uninitialized)"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The code-behind file contains the handler for the `ValueChanged` event:

```

public partial class BasicSliderXamlPage : ContentPage
{
    public BasicSliderXamlPage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        double value = args.NewValue;
        rotatingLabel.Rotation = value;
        displayLabel.Text = String.Format("The Slider value is {0}", value);
    }
}

```

It's also possible for the event handler to obtain the `Slider` that is firing the event through the `sender` argument. The `value` property contains the current value:

```
double value = ((Slider)sender).Value;
```

If the `slider` object were given a name in the XAML file with an `x:Name` attribute (for example, "slider"), then the event handler could reference that object directly:

```
double value = slider.Value;
```

## Data binding the Slider

The [Basic Slider Bindings](#) page shows how to write a nearly equivalent program that eliminates the `Value` event handler by using [Data Binding](#):

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SliderDemos.BasicSliderBindingsPage"
    Title="Basic Slider Bindings"
    Padding="10, 0">

    <StackLayout>
        <Label Text="ROTATING TEXT"
            Rotation="{Binding Source={x:Reference slider},
                Path=Value}"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
            Maximum="360" />

        <Label x:Name="displayLabel"
            Text="{Binding Source={x:Reference slider},
                Path=Value,
                StringFormat='The Slider value is {0:F0}'}"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

The `Rotation` property of the first `Label` is bound to the `Value` property of the `Slider`, as is the `Text` property of the second `Label` with a `StringFormat` specification. The [Basic Slider Bindings](#) page functions a little differently from the two previous pages: When the page first appears, the second `Label` displays the text string with the value. This is a benefit of using data binding. To display text without data binding, you'd need to

specifically initialize the `Text` property of the `Label` or simulate a firing of the `ValueChanged` event by calling the event handler from the class constructor.

## Precautions

The value of the `Minimum` property must always be less than the value of the `Maximum` property. The following code snippet causes the `Slider` to raise an exception:

```
// Throws an exception!
Slider slider = new Slider
{
    Minimum = 10,
    Maximum = 20
};
```

The C# compiler generates code that sets these two properties in sequence, and when the `Minimum` property is set to 10, it is greater than the default `Maximum` value of 1. You can avoid the exception in this case by setting the `Maximum` property first:

```
Slider slider = new Slider
{
    Maximum = 20,
    Minimum = 10
};
```

Setting `Maximum` to 20 is not a problem because it is greater than the default `Minimum` value of 0. When `Minimum` is set, the value is less than the `Maximum` value of 20.

The same problem exists in XAML. Set the properties in an order that ensures that `Maximum` is always greater than `Minimum`:

```
<Slider Maximum="20"
        Minimum="10" ... />
```

You can set the `Minimum` and `Maximum` values to negative numbers, but only in an order where `Minimum` is always less than `Maximum`:

```
<Slider Minimum="-20"
        Maximum="-10" ... />
```

The `Value` property is always greater than or equal to the `Minimum` value and less than or equal to `Maximum`. If `Value` is set to a value outside that range, the value will be coerced to lie within the range, but no exception is raised. For example, this code will *not* raise an exception:

```
Slider slider = new Slider
{
    Value = 10
};
```

Instead, the `Value` property is coerced to the `Maximum` value of 1.

Here's a code snippet shown above:

```
Slider slider = new Slider
{
    Maximum = 20,
    Minimum = 10
};
```

When `Minimum` is set to 10, then `Value` is also set to 10.

If a `ValueChanged` event handler has been attached at the time that the `Value` property is coerced to something other than its default value of 0, then a `ValueChanged` event is fired. Here's a snippet of XAML:

```
<Slider ValueChanged="OnSliderValueChanged"
        Maximum="20"
        Minimum="10" />
```

When `Minimum` is set to 10, `Value` is also set to 10, and the `ValueChanged` event is fired. This might occur before the rest of the page has been constructed, and the handler might attempt to reference other elements on the page that have not yet been created. You might want to add some code to the `ValueChanged` handler that checks for `null` values of other elements on the page. Or, you can set the `ValueChanged` event handler after the `Slider` values have been initialized.

## Platform implementation differences

The screenshots shown earlier display the value of the `Slider` with a different number of decimal points. This relates to how the `Slider` is implemented on the Android and UWP platforms.

### The Android implementation

The Android implementation of `Slider` is based on the Android `SeekBar` and always sets the `Max` property to 1000. This means that the `Slider` on Android has only 1,001 discrete values. If you set the `Slider` to have a `Minimum` of 0 and a `Maximum` of 5000, then as the `Slider` is manipulated, the `Value` property has values of 0, 5, 10, 15, and so forth.

### The UWP implementation

The UWP implementation of `Slider` is based on the UWP `Slider` control. The `StepFrequency` property of the UWP `Slider` is set to the difference of the `Maximum` and `Minimum` properties divided by 10, but not greater than 1.

For example, for the default range of 0 to 1, the `StepFrequency` property is set to 0.1. As the `Slider` is manipulated, the `Value` property is restricted to 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. (This is evident in the last page in the [SliderDemos](#) sample.) When the difference between the `Maximum` and `Minimum` properties is 10 or greater, then `StepFrequency` is set to 1, and the `Value` property has integral values.

### The StepSlider solution

A more versatile `StepSlider` is discussed in [Chapter 27. Custom renderers](#) of the book *Creating Mobile Apps with Xamarin.Forms*. The `StepSlider` is similar to `Slider` but adds a `Steps` property to specify the number of values between `Minimum` and `Maximum`.

## Sliders for color selection

The final two pages in the [SliderDemos](#) sample both use three `Slider` instances for color selection. The first page handles all the interactions in the code-behind file, while the second page shows how to use data binding with a ViewModel.

### Handling Sliders in the code-behind file

The **RGB Color Sliders** page instantiates a `BoxView` to display a color, three `Slider` instances to select the red, green, and blue components of the color, and three `Label` elements for displaying those color values:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="SliderDemos.RgbColorSlidersPage"
    Title="RGB Color Sliders">
<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Slider">
            <Setter Property="Maximum" Value="255" />
        </Style>

        <Style TargetType="Label">
            <Setter Property="HorizontalTextAlignment" Value="Center" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<StackLayout Margin="10">
    <BoxView x:Name="boxView"
        Color="Black"
        VerticalOptions="FillAndExpand" />

    <Slider x:Name="redSlider"
        ValueChanged="OnSliderValueChanged" />

    <Label x:Name="redLabel" />

    <Slider x:Name="greenSlider"
        ValueChanged="OnSliderValueChanged" />

    <Label x:Name="greenLabel" />

    <Slider x:Name="blueSlider"
        ValueChanged="OnSliderValueChanged" />

    <Label x:Name="blueLabel" />
</StackLayout>
</ContentPage>
```

A `Style` gives all three `Slider` elements a range of 0 to 255. The `Slider` elements share the same `ValueChanged` handler, which is implemented in the code-behind file:

```

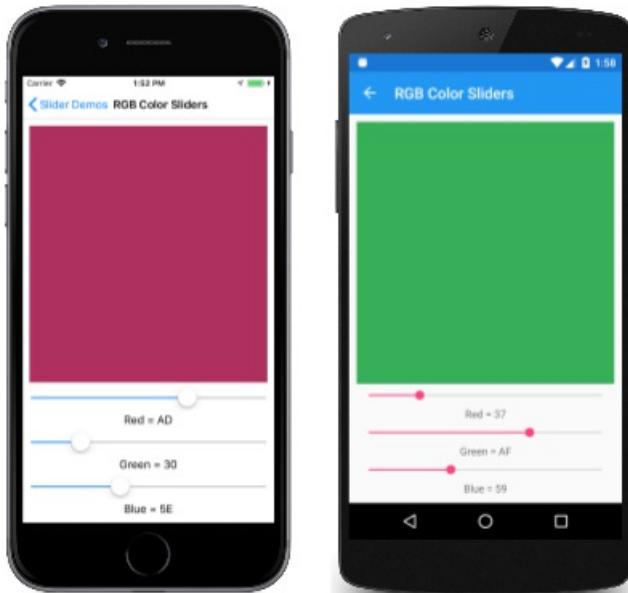
public partial class RgbColorSlidersPage : ContentPage
{
    public RgbColorSlidersPage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (sender == redSlider)
        {
            redLabel.Text = String.Format("Red = {0:X2}", (int)args.NewValue);
        }
        else if (sender == greenSlider)
        {
            greenLabel.Text = String.Format("Green = {0:X2}", (int)args.NewValue);
        }
        else if (sender == blueSlider)
        {
            blueLabel.Text = String.Format("Blue = {0:X2}", (int)args.NewValue);
        }

        boxView.Color = Color.FromRgb((int)redSlider.Value,
                                      (int)greenSlider.Value,
                                      (int)blueSlider.Value);
    }
}

```

The first section sets the `Text` property of one of the `Label` instances to a short text string indicating the value of the `Slider` in hexadecimal. Then, all three `Slider` instances are accessed to create a `color` value from the RGB components:



### Binding the Slider to a ViewModel

The **HSL Color Sliders** page shows how to use a ViewModel to perform the calculations used to create a `color` value from hue, saturation, and luminosity values. Like all ViewModels, the `HslColorViewModel` class implements the `INotifyPropertyChanged` interface, and fires a `PropertyChanged` event whenever one of the properties changes:

```

public class HslColorViewModel : INotifyPropertyChanged
{
    Color color;

    public event PropertyChangedEventHandler PropertyChanged;

```

```

public double Hue
{
    set
    {
        if (color.Hue != value)
        {
            Color = Color.FromHsla(value, color.Saturation, color.Luminosity);
        }
    }
    get
    {
        return color.Hue;
    }
}

public double Saturation
{
    set
    {
        if (color.Saturation != value)
        {
            Color = Color.FromHsla(color.Hue, value, color.Luminosity);
        }
    }
    get
    {
        return color.Saturation;
    }
}

public double Luminosity
{
    set
    {
        if (color.Luminosity != value)
        {
            Color = Color.FromHsla(color.Hue, color.Saturation, value);
        }
    }
    get
    {
        return color.Luminosity;
    }
}

public Color Color
{
    set
    {
        if (color != value)
        {
            color = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Hue"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Saturation"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Luminosity"));
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));
        }
    }
    get
    {
        return color;
    }
}
}

```

ViewModels and the `IPropertyChanged` interface are discussed in the article [Data Binding](#).

The **HslColorSlidersPage.xaml** file instantiates the `HslColorViewModel` and sets it to the page's `BindingContext` property. This allows all the elements in the XAML file to bind to properties in the ViewModel:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:SliderDemos"
    x:Class="SliderDemos.HslColorSlidersPage"
    Title="HSL Color Sliders">

    <ContentPage.BindingContext>
        <local:HslColorViewModel Color="Chocolate" />
    </ContentPage.BindingContext>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

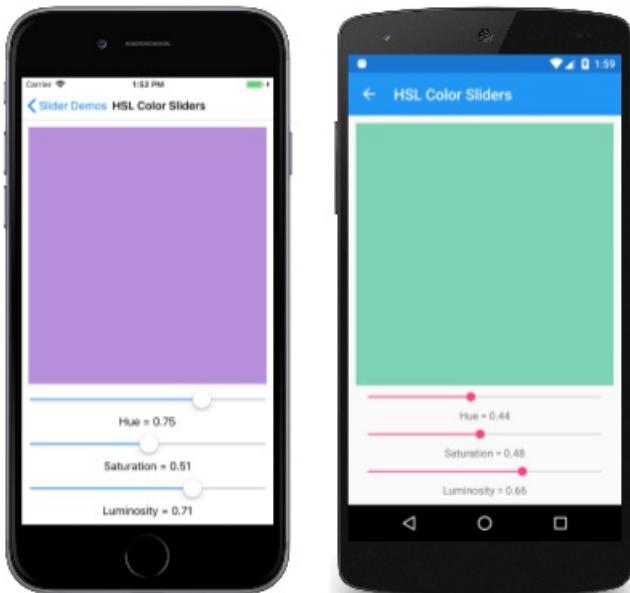
    <StackLayout Margin="10">
        <BoxView Color="{Binding Color}"
            VerticalOptions="FillAndExpand" />

        <Slider Value="{Binding Hue}" />
        <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />

        <Slider Value="{Binding Saturation}" />
        <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />

        <Slider Value="{Binding Luminosity}" />
        <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
    </StackLayout>
</ContentPage>
```

As the `Slider` elements are manipulated, the `BoxView` and `Label` elements are updated from the ViewModel:



The `StringFormat` component of the `Binding` markup extension is set for a format of "F2" to display two decimal places. (String formatting in data bindings is discussed in the article [String Formatting](#).) However, the UWP version of the program is limited to values of 0, 0.1, 0.2, ... 0.9, and 1.0. This is a direct result of the implementation of the UWP `Slider` as described above in the section [Platform implementation differences](#).

## Related Links

- [Slider Demos sample](#)
- [Slider API](#)

# Xamarin.Forms Stepper

10/17/2018 • 7 minutes to read • [Edit Online](#)

Use a Stepper for selecting a numeric value from a range of values.

The Xamarin.Forms `Stepper` consists of two buttons labeled with minus and plus signs. These buttons can be manipulated by the user to incrementally select a `double` value from a range of values.

The `Stepper` defines four properties of type `double`:

- `Increment` is the amount to change the selected value by, with a default value of 1.
- `Minimum` is the minimum of the range, with a default value of 0.
- `Maximum` is the maximum of the range, with a default value of 100.
- `Value` is the stepper's value, which can range between `Minimum` and `Maximum` and has a default value of 0.

All of these properties are backed by `BindableProperty` objects. The `Value` property has a default binding mode of `BindingMode.TwoWay`, which means that it's suitable as a binding source in an application that uses the [Model-View-ViewModel \(MVVM\)](#) architecture.

## WARNING

Internally, the `Stepper` ensures that `Minimum` is less than `Maximum`. If `Minimum` or `Maximum` are ever set so that `Minimum` is not less than `Maximum`, an exception is raised. For more information on setting the `Minimum` and `Maximum` properties, see [Precautions](#) section.

The `Stepper` coerces the `Value` property so that it is between `Minimum` and `Maximum`, inclusive. If the `Minimum` property is set to a value greater than the `Value` property, the `Stepper` sets the `Value` property to `Minimum`. Similarly, if `Maximum` is set to a value less than `Value`, then `Stepper` sets the `Value` property to `Maximum`.

`Stepper` defines a `ValueChanged` event that is fired when the `Value` changes, either through user manipulation of the `Stepper` or when the application sets the `Value` property directly. A `ValueChanged` event is also fired when the `Value` property is coerced as described in the previous paragraph.

The `ValueChangedEventArgs` object that accompanies the `ValueChanged` event has two properties, both of type `double`: `OldValue` and `NewValue`. At the time the event is fired, the value of `NewValue` is the same as the `Value` property of the `Stepper` object.

## Basic Stepper code and markup

The [StepperDemos](#) sample contains three pages that are functionally identical, but are implemented in different ways. The first page uses only C# code, the second uses XAML with an event handler in code, and third is able to avoid the event handler by using data binding in the XAML file.

### Creating a Stepper in code

The [Basic Stepper Code](#) page in the [StepperDemos](#) sample shows how to create a `Stepper` and two `Label` objects in code:

```

public class BasicStepperCodePage : ContentPage
{
    public BasicStepperCodePage()
    {
        Label rotationLabel = new Label
        {
            Text = "ROTATING TEXT",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Label displayLabel = new Label
        {
            Text = "(uninitialized)",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

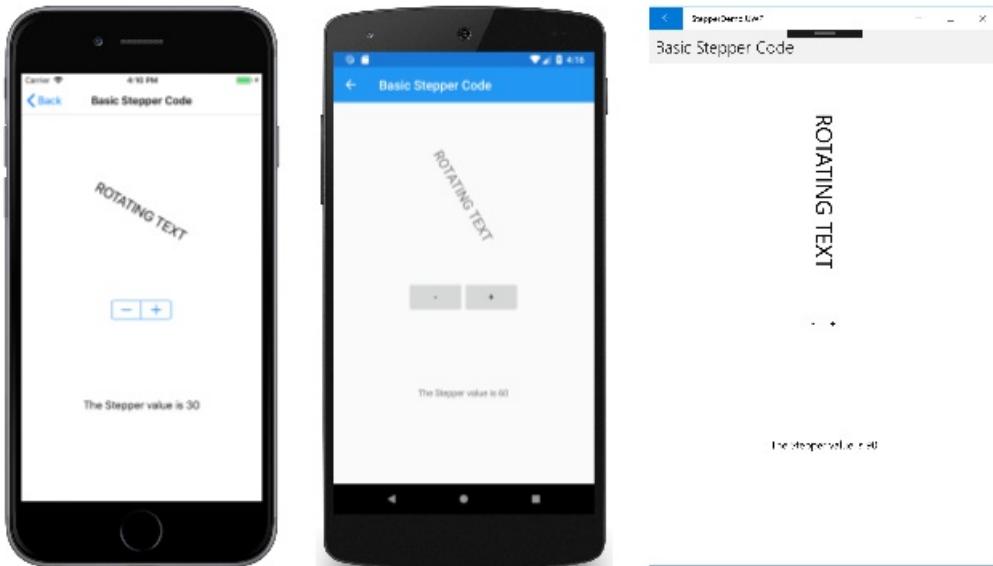
        Stepper stepper = new Stepper
        {
            Maximum = 360,
            Increment = 30,
            HorizontalOptions = LayoutOptions.Center
        };
        stepper.ValueChanged += (sender, e) =>
        {
            rotationLabel.Rotation = stepper.Value;
            displayLabel.Text = string.Format("The Stepper value is {0}", e.NewValue);
        };
    }

    Title = "Basic Stepper Code";
    Content = new StackLayout
    {
        Margin = new Thickness(20),
        Children = { rotationLabel, stepper, displayLabel }
    };
}

```

The `Stepper` is initialized to have a `Maximum` property of 360, and an `Increment` property of 30. Manipulating the `Stepper` changes the selected value incrementally between `Minimum` to `Maximum` based on the value of the `Increment` property. The `ValueChanged` handler of the `Stepper` uses the `Value` property of the `stepper` object to set the `Rotation` property of the first `Label` and uses the `string.Format` method with the `NewValue` property of the event arguments to set the `Text` property of the second `Label`. These two approaches to obtain the current value of the `Stepper` are interchangeable.

The following screenshots show the **Basic Stepper Code** page:



The second `Label` displays the text "(uninitialized)" until the `Stepper` is manipulated, which causes the first `ValueChanged` event to be fired.

### Creating a Stepper in XAML

The **Basic Stepper XAML** page is functionally the same as **Basic Stepper Code** but implemented mostly in XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StepperDemo.BasicStepperXAMLPage"
    Title="Basic Stepper XAML">
    <StackLayout Margin="20">
        <Label x:Name="_rotatingLabel"
            Text="ROTATING TEXT"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
        <Stepper Maximum="360"
            Increment="30"
            HorizontalOptions="Center"
            ValueChanged="OnStepperValueChanged" />
        <Label x:Name="_displayLabel"
            Text="(uninitialized)"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The code-behind file contains the handler for the `ValueChanged` event:

```

public partial class BasicStepperXAMLPage : ContentPage
{
    public BasicStepperXAMLPage()
    {
        InitializeComponent();
    }

    void OnStepperValueChanged(object sender, ValueChangedEventArgs e)
    {
        double value = e.NewValue;
        _rotatingLabel.Rotation = value;
        _displayLabel.Text = string.Format("The Stepper value is {0}", value);
    }
}

```

It's also possible for the event handler to obtain the `Stepper` that is firing the event through the `sender` argument. The `Value` property contains the current value:

```
double value = ((Stepper)sender).Value;
```

If the `Stepper` object were given a name in the XAML file with an `x:Name` attribute (for example, "stepper"), then the event handler could reference that object directly:

```
double value = stepper.Value;
```

## Data binding the Stepper

The **Basic Stepper Bindings** page shows how to write a nearly equivalent application that eliminates the `Value` event handler by using [Data Binding](#):

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StepperDemo.BasicStepperBindingsPage"
    Title="Basic Stepper Bindings">
    <StackLayout Margin="20">
        <Label Text="ROTATING TEXT"
            Rotation="{Binding Source={x:Reference _stepper}, Path=Value}"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
        <Stepper x:Name="_stepper"
            Maximum="360"
            Increment="30"
            HorizontalOptions="Center" />
        <Label Text="{Binding Source={x:Reference _stepper}, Path=Value, StringFormat='The Stepper value is {0:F0}'}"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>

```

The `Rotation` property of the first `Label` is bound to the `Value` property of the `Stepper`, as is the `Text` property of the second `Label` with a `StringFormat` specification. The **Basic Stepper Bindings** page functions a little differently from the two previous pages: When the page first appears, the second `Label` displays the text string with the value. This is a benefit of using data binding. To display text without data binding, you'd need to specifically initialize the `Text` property of the `Label` or simulate a firing of the `ValueChanged` event by calling the event handler from the class constructor.

## Precautions

The value of the `Minimum` property must always be less than the value of the `Maximum` property. The following code snippet causes the `Stepper` to raise an exception:

```
// Throws an exception!
Stepper stepper = new Stepper
{
    Minimum = 180,
    Maximum = 360
};
```

The C# compiler generates code that sets these two properties in sequence, and when the `Minimum` property is set to 180, it is greater than the default `Maximum` value of 100. You can avoid the exception in this case by setting the `Maximum` property first:

```
Stepper stepper = new Stepper
{
    Maximum = 360,
    Minimum = 180
};
```

Setting `Maximum` to 360 is not a problem because it is greater than the default `Minimum` value of 0. When `Minimum` is set, the value is less than the `Maximum` value of 360.

The same problem exists in XAML. Set the properties in an order that ensures that `Maximum` is always greater than `Minimum`:

```
<Stepper Maximum="360"
         Minimum="180" ... />
```

You can set the `Minimum` and `Maximum` values to negative numbers, but only in an order where `Minimum` is always less than `Maximum`:

```
<Stepper Minimum="-360"
         Maximum="-180" ... />
```

The `Value` property is always greater than or equal to the `Minimum` value and less than or equal to `Maximum`. If `Value` is set to a value outside that range, the value will be coerced to lie within the range, but no exception is raised. For example, this code will *not* raise an exception:

```
Stepper stepper = new Stepper
{
    Value = 180
};
```

Instead, the `Value` property is coerced to the `Maximum` value of 100.

Here's a code snippet shown above:

```
Stepper stepper = new Stepper  
{  
    Maximum = 360,  
    Minimum = 180  
};
```

When `Minimum` is set to 180, then `Value` is also set to 180.

If a `ValueChanged` event handler has been attached at the time that the `Value` property is coerced to something other than its default value of 0, then a `ValueChanged` event is fired. Here's a snippet of XAML:

```
<Stepper ValueChanged="OnStepperValueChanged"  
        Maximum="360"  
        Minimum="180" />
```

When `Minimum` is set to 180, `Value` is also set to 180, and the `ValueChanged` event is fired. This might occur before the rest of the page has been constructed, and the handler might attempt to reference other elements on the page that have not yet been created. You might want to add some code to the `ValueChanged` handler that checks for `null` values of other elements on the page. Or, you can set the `ValueChanged` event handler after the `Stepper` values have been initialized.

## Related Links

- [Stepper Demos sample](#)
- [Stepper API](#)

# Styling Xamarin.Forms Apps

7/12/2018 • 2 minutes to read • [Edit Online](#)

## Styling Xamarin.Forms Apps using XAML Styles

Styling a Xamarin.Forms app is traditionally accomplished by using the `Style` class to group a collection of property values into one object that can then be applied to multiple visual element instances. This helps to reduce repetitive markup, and allows an apps appearance to be more easily changed.

## Styling Xamarin.Forms Apps using Cascading Style Sheets

Xamarin.Forms supports styling visual elements using Cascading Style Sheets (CSS). A style sheet consists of a list of rules, with each rule consisting of one or more selectors, and a declaration block.

# Styling Xamarin.Forms Apps using XAML Styles

7/12/2018 • 2 minutes to read • [Edit Online](#)

## Introduction

Xamarin.Forms applications often contain multiple controls that have an identical appearance. Setting the appearance of each individual control can be repetitive and error prone. Instead, styles can be created that customize control appearance by grouping and settings properties available on the control type.

## Explicit Styles

An *explicit* style is one that is selectively applied to controls by setting their `Style` properties.

## Implicit Styles

An *implicit* style is one that's used by all controls of the same `TargetType`, without requiring each control to reference the style.

## Global Styles

Styles can be made available globally by adding them to the application's `ResourceDictionary`. This helps to avoid duplication of styles across pages or controls.

## Style Inheritance

Styles can inherit from other styles to reduce duplication and enable reuse.

## Dynamic Styles

Styles do not respond to property changes, and remain unchanged for the duration of an application. However, applications can respond to style changes dynamically at runtime by using dynamic resources.

## Device Styles

Xamarin.Forms includes six *dynamic* styles, known as *device* styles, in the `Devices.Styles` class. All six styles can be applied to `Label` instances only.

# Introduction to Xamarin.Forms Styles

7/12/2018 • 4 minutes to read • [Edit Online](#)

Styles allow the appearance of visual elements to be customized. Styles are defined for a specific type and contain values for the properties available on that type.

Xamarin.Forms applications often contain multiple controls that have an identical appearance. For example, an application may have multiple `Label` instances that have the same font options and layout options, as shown in the following XAML code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Styles.NoStylesPage"
    Title="No Styles"
    Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                FontSize="Large" />
            <Label Text="are not"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                FontSize="Large" />
            <Label Text="using styles"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                FontSize="Large" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The following code example shows the equivalent page created in C#:

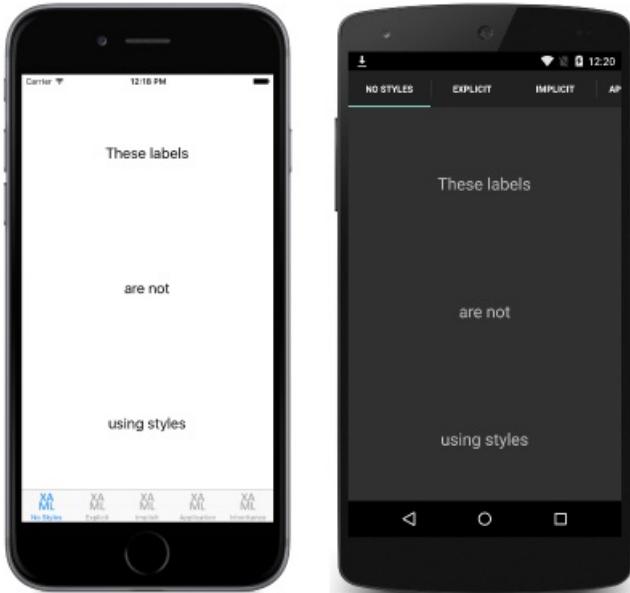
```

public class NoStylesPageCS : ContentPage
{
    public NoStylesPageCS ()
    {
        Title = "No Styles";
        Icon = "csharp.png";
        Padding = new Thickness (0, 20, 0, 0);

        Content = new StackLayout {
            Children = {
                new Label {
                    Text = "These labels",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                },
                new Label {
                    Text = "are not",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                },
                new Label {
                    Text = "using styles",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                }
            }
        };
    }
}

```

Each `Label` instance has identical property values for controlling the appearance of the text displayed by the `Label`. This results in the appearance shown in the following screenshots:



Setting the appearance of each individual control can be repetitive and error prone. Instead, a style can be created that defines the appearance, and then applied to the required controls.

## Creating a Style

The `Style` class groups a collection of property values into one object that can then be applied to multiple visual element instances. This helps to reduce repetitive markup, and allows an application's appearance to be more easily

changed.

Although styles were designed primarily for XAML-based applications, they can also be created in C#:

- `Style` instances created in XAML are typically defined in a `ResourceDictionary` that's assigned to the `Resources` collection of a control, page, or to the `Resources` collection of the application.
- `Style` instances created in C# are typically defined in the page's class, or in a class that can be globally accessed.

Choosing where to define a `Style` impacts where it can be used:

- `Style` instances defined at the control level can only be applied to the control and to its children.
- `Style` instances defined at the page level can only be applied to the page and to its children.
- `Style` instances defined at the application level can be applied throughout the application.

Each `Style` instance contains a collection of one or more `Setter` objects, with each `Setter` having a `Property` and a `Value`. The `Property` is the name of the bindable property of the element the style is applied to, and the `Value` is the value that is applied to the property.

Each `Style` instance can be *explicit*, or *implicit*:

- An *explicit* `Style` instance is defined by specifying a `TargetType` and an `x:Key` value, and by setting the target element's `Style` property to the `x:Key` reference. For more information about *explicit* styles, see [Explicit Styles](#).
- An *implicit* `Style` instance is defined by specifying only a `TargetType`. The `Style` instance will then automatically be applied to all elements of that type. Note that subclasses of the `TargetType` do not automatically have the `Style` applied. For more information about *implicit* styles, see [Implicit Styles](#).

When creating a `Style`, the `TargetType` property is always required. The following code example shows an *explicit* style (note the `x:Key`) created in XAML:

```
<Style x:Key="labelStyle" TargetType="Label">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
    <Setter Property="FontSize" Value="Large" />
</Style>
```

To apply a `Style`, the target object must be a `visualElement` that matches the `TargetType` property value of the `Style`, as shown in the following XAML code example:

```
<Label Text="Demonstrating an explicit style" Style="{StaticResource labelStyle}" />
```

Styles lower in the view hierarchy take precedence over those defined higher up. For example, setting a `Style` that sets `Label.TextColor` to `Red` at the application level will be overridden by a page level style that sets `Label.TextColor` to `Green`. Similarly, a page level style will be overridden by a control level style. In addition, if `Label.TextColor` is set directly on a control property, this takes precedence over any styles.

The articles in this section demonstrate and explain how to create and apply *explicit* and *implicit* styles, how to create global styles, style inheritance, how to respond to style changes at runtime, and how to use the in-built styles included in Xamarin.Forms.

## NOTE

### What is StyleId?

Prior to Xamarin.Forms 2.2, the `StyleId` property was used to identify individual elements in an application for identification in UI testing, and in theme engines such as Pixate. However, Xamarin.Forms 2.2 has introduced the `AutomationId` property, which has superseded the `StyleId` property. For more information, see [Automate Xamarin.Forms testing with Xamarin.UITest and Test Cloud](#).

## Summary

Xamarin.Forms applications often contain multiple controls that have an identical appearance. Setting the appearance of each individual control can be repetitive and error prone. Instead, styles can be created that customize control appearance by grouping and settings properties available on the control type.

## Related Links

- [XAML Markup Extensions](#)
- [Style](#)
- [Setter](#)

# Explicit Styles in Xamarin.Forms

7/12/2018 • 4 minutes to read • [Edit Online](#)

An *explicit style* is one that is selectively applied to controls by setting their `Style` properties.

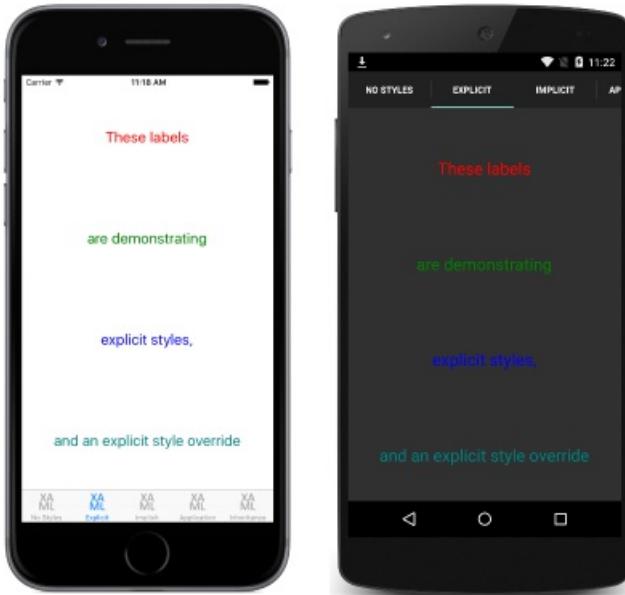
## Creating an Explicit Style in XAML

To declare a `Style` at the page level, a `ResourceDictionary` must be added to the page and then one or more `Style` declarations can be included in the `ResourceDictionary`. A `style` is made *explicit* by giving its declaration an `x:Key` attribute, which gives it a descriptive key in the `ResourceDictionary`. *Explicit* styles must then be applied to specific visual elements by setting their `Style` properties.

The following code example shows *explicit* styles declared in XAML in a page's `ResourceDictionary` and applied to the page's `Label` instances:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ExplicitStylesPage" Title="Explicit"
Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="labelRedStyle" TargetType="Label">
                <Setter Property="HorizontalOptions"
                    Value="Center" />
                <Setter Property="VerticalOptions"
                    Value="CenterAndExpand" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
            <Style x:Key="labelGreenStyle" TargetType="Label">
                ...
                <Setter Property="TextColor" Value="Green" />
            </Style>
            <Style x:Key="labelBlueStyle" TargetType="Label">
                ...
                <Setter Property="TextColor" Value="Blue" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                Style="{StaticResource labelRedStyle}" />
            <Label Text="are demonstrating"
                Style="{StaticResource labelGreenStyle}" />
            <Label Text="explicit styles,"
                Style="{StaticResource labelBlueStyle}" />
            <Label Text="and an explicit style override"
                Style="{StaticResource labelBlueStyle}"
                TextColor="Teal" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The `ResourceDictionary` defines three *explicit* styles that are applied to the page's `Label` instances. Each `Style` is used to display text in a different color, while also setting the font size and horizontal and vertical layout options. Each `style` is applied to a different `Label` by setting its `Style` properties using the `StaticResource` markup extension. This results in the appearance shown in the following screenshots:



In addition, the final `Label` has a `Style` applied to it, but also overrides the `TextColor` property to a different `Color` value.

### Creating an Explicit Style at the Control Level

In addition to creating *explicit* styles at the page level, they can also be created at the control level, as shown in the following code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ExplicitStylesPage" Title="Explicit"
Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="labelRedStyle" TargetType="Label">
                        ...
                    </Style>
                    ...
                </ResourceDictionary>
            </StackLayout.Resources>
            <Label Text="These labels" Style="{StaticResource labelRedStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

In this example, the *explicit* `Style` instances are assigned to the `Resources` collection of the `StackLayout` control. The styles can then be applied to the control and its children.

For information about creating styles in an application's `ResourceDictionary`, see [Global Styles](#).

## Creating an Explicit Style in C#

`Style` instances can be added to a page's `Resources` collection in C# by creating a new `ResourceDictionary`, and then by adding the `Style` instances to the `ResourceDictionary`, as shown in the following code example:

```

public class ExplicitStylesPageCS : ContentPage
{
    public ExplicitStylesPageCS ()
    {
        var labelRedStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Red }
            }
        };
        var labelGreenStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Green }
            }
        };
        var labelBlueStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Blue }
            }
        };

        Resources = new ResourceDictionary ();
        Resources.Add ("labelRedStyle", labelRedStyle);
        Resources.Add ("labelGreenStyle", labelGreenStyle);
        Resources.Add ("labelBlueStyle", labelBlueStyle);
        ...

        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels",
                    Style = (Style)Resources ["labelRedStyle"] },
                new Label { Text = "are demonstrating",
                    Style = (Style)Resources ["labelGreenStyle"] },
                new Label { Text = "explicit styles",
                    Style = (Style)Resources ["labelBlueStyle"] },
                new Label { Text = "and an explicit style override",
                    Style = (Style)Resources ["labelBlueStyle"], TextColor = Color.Teal }
            }
        };
    }
}

```

The constructor defines three *explicit* styles that are applied to the page's `Label` instances. Each *explicit* `Style` is added to the `ResourceDictionary` using the `Add` method, specifying a `key` string to refer to the `style` instance. Each `style` is applied to a different `Label` by setting their `Style` properties.

However, there is no advantage to using a `ResourceDictionary` here. Instead, `Style` instances can be assigned directly to the `Style` properties of the required visual elements, and the `ResourceDictionary` can be removed, as shown in the following code example:

```

public class ExplicitStylesPageCS : ContentPage
{
    public ExplicitStylesPageCS ()
    {
        var labelRedStyle = new Style (typeof(Label)) {
            ...
        };
        var labelGreenStyle = new Style (typeof(Label)) {
            ...
        };
        var labelBlueStyle = new Style (typeof(Label)) {
            ...
        };
        ...
        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels", Style = labelRedStyle },
                new Label { Text = "are demonstrating", Style = labelGreenStyle },
                new Label { Text = "explicit styles,", Style = labelBlueStyle },
                new Label { Text = "and an explicit style override", Style = labelBlueStyle,
                    TextColor = Color.Teal }
            }
        };
    }
}

```

The constructor defines three *explicit* styles that are applied to the page's `Label` instances. Each `style` is used to display text in a different color, while also setting the font size and horizontal and vertical layout options. Each `Style` is applied to a different `Label` by setting its `Style` properties. In addition, the final `Label` has a `Style` applied to it, but also overrides the `TextColor` property to a different `Color` value.

## Summary

A `Style` is made *explicit* by giving its declaration an `x:Key` attribute, and then selectively applying it to controls by setting their `Style` properties.

## Related Links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Implicit Styles in Xamarin.Forms

7/12/2018 • 3 minutes to read • [Edit Online](#)

An *implicit style* is one that's used by all controls of the same `TargetType`, without requiring each control to reference the style.

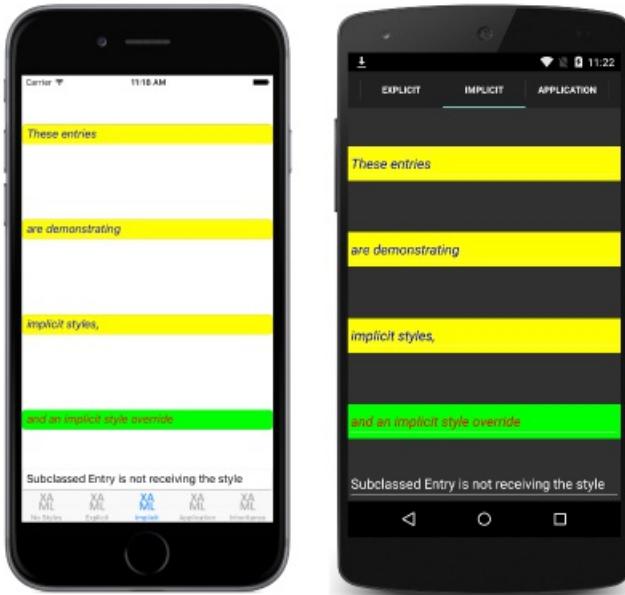
## Creating an Implicit Style in XAML

To declare a `Style` at the page level, a `ResourceDictionary` must be added to the page and then one or more `Style` declarations can be included in the `ResourceDictionary`. A `Style` is made *implicit* by not specifying an `x:Key` attribute. The style will then be applied to visual elements that match the `TargetType` exactly, but not to elements that are derived from the `TargetType` value.

The following code example shows an *implicit* style declared in XAML in a page's `ResourceDictionary`, and applied to the page's `Entry` instances:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Styles;assembly=Styles"
    x:Class="Styles.ImplicitStylesPage" Title="Implicit" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Setter Property="HorizontalOptions" Value="Fill" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BackgroundColor" Value="Yellow" />
                <Setter Property="FontAttributes" Value="Italic" />
                <Setter Property="TextColor" Value="Blue" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Entry Text="These entries" />
            <Entry Text="are demonstrating" />
            <Entry Text="implicit styles," />
            <Entry Text="and an implicit style override" BackgroundColor="Lime" TextColor="Red" />
            <local:CustomEntry Text="Subclassed Entry is not receiving the style" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The `ResourceDictionary` defines a single *implicit* style that's applied to the page's `Entry` instances. The `Style` is used to display blue text on a yellow background, while also setting other appearance options. The `Style` is added to the page's `ResourceDictionary` without specifying an `x:Key` attribute. Therefore, the `Style` is applied to all the `Entry` instances implicitly as they match the `TargetType` property of the `Style` exactly. However, the `Style` is not applied to the `CustomEntry` instance, which is a subclassed `Entry`. This results in the appearance shown in the following screenshots:



In addition, the fourth `Entry` overrides the `BackgroundColor` and `TextColor` properties of the implicit style to different `Color` values.

### Creating an Implicit Style at the Control Level

In addition to creating *implicit* styles at the page level, they can also be created at the control level, as shown in the following code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Styles;assembly=Styles"
    x:Class="Styles.ImplicitStylesPage" Title="Implicit" Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style TargetType="Entry">
                        <Setter Property="HorizontalOptions" Value="Fill" />
                        ...
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            <Entry Text="These entries" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

In this example, the *implicit* `Style` is assigned to the `Resources` collection of the `StackLayout` control. The *implicit* style can then be applied to the control and its children.

For information about creating styles in an application's `ResourceDictionary`, see [Global Styles](#).

## Creating an Implicit Style in C#

`Style` instances can be added to a page's `Resources` collection in C# by creating a new `ResourceDictionary`, and then by adding the `Style` instances to the `ResourceDictionary`, as shown in the following code example:

```

public class ImplicitStylesPageCS : ContentPage
{
    public ImplicitStylesPageCS ()
    {
        var entryStyle = new Style (typeof(Entry)) {
            Setters = {
                ...
                new Setter { Property = Entry.TextColorProperty, Value = Color.Blue }
            }
        };

        ...
        Resources = new ResourceDictionary ();
        Resources.Add (entryStyle);

        Content = new StackLayout {
            Children = {
                new Entry { Text = "These entries" },
                new Entry { Text = "are demonstrating" },
                new Entry { Text = "implicit styles," },
                new Entry { Text = "and an implicit style override", BackgroundColor = Color.Lime, TextColor =
Color.Red },
                new CustomEntry { Text = "Subclassed Entry is not receiving the style" }
            }
        };
    }
}

```

The constructor defines a single *implicit* style that's applied to the page's `Entry` instances. The `Style` is used to display blue text on a yellow background, while also setting other appearance options. The `Style` is added to the page's `ResourceDictionary` without specifying a `key` string. Therefore, the `Style` is applied to all the `Entry` instances implicitly as they match the `TargetType` property of the `Style` exactly. However, the `Style` is not applied to the `CustomEntry` instance, which is a subclassed `Entry`.

## Summary

An *implicit* style is one that's used by all visual elements of the same `TargetType`, without requiring each control to reference the style. A `Style` is made *implicit* by not specifying an `x:Key` attribute. Instead, the `x:Key` attribute will automatically become the value of the `TargetType` property.

## Related Links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Global Styles in Xamarin.Forms

7/12/2018 • 3 minutes to read • [Edit Online](#)

Styles can be made available globally by adding them to the application's resource dictionary. This helps to avoid duplication of styles across pages or controls.

## Creating a Global Style in XAML

By default, all Xamarin.Forms applications created from a template use the **App** class to implement the **Application** subclass. To declare a **Style** at the application level, in the application's **ResourceDictionary** using XAML, the default **App** class must be replaced with a XAML **App** class and associated code-behind. For more information, see [Working with the App Class](#).

The following code example shows a **Style** declared at the application level:

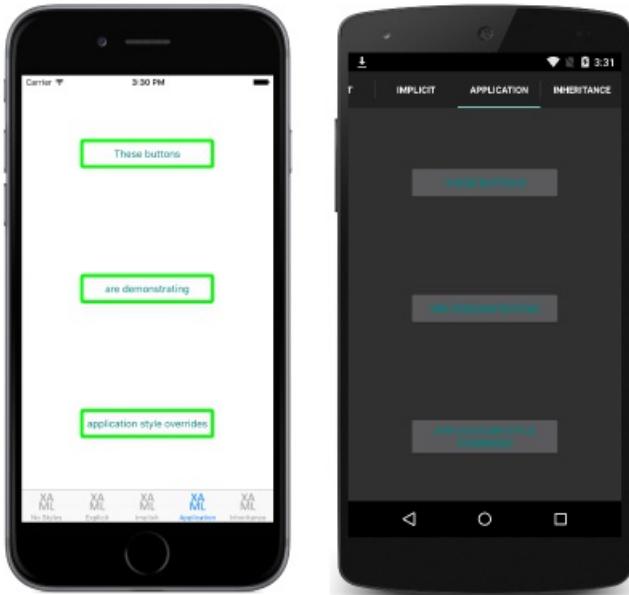
```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.App">
  <Application.Resources>
    <ResourceDictionary>
      <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="HorizontalOptions" Value="Center" />
        <Setter Property="VerticalOptions" Value="CenterAndExpand" />
        <Setter Property="BorderColor" Value="Lime" />
        <Setter Property="BorderRadius" Value="5" />
        <Setter Property="BorderWidth" Value="5" />
        <Setter Property="WidthRequest" Value="200" />
        <Setter Property="TextColor" Value="Teal" />
      </Style>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

This **ResourceDictionary** defines a single *explicit* style, **buttonStyle**, which will be used to set the appearance of **Button** instances. However, global styles can be *explicit* or *implicit*.

The following code example shows a XAML page applying the **buttonStyle** to the page's **Button** instances:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ApplicationStylesPage"
  Title="Application" Icon="xaml.png">
  <ContentPage.Content>
    <StackLayout Padding="0,20,0,0">
      <Button Text="These buttons" Style="{StaticResource buttonStyle}" />
      <Button Text="are demonstrating" Style="{StaticResource buttonStyle}" />
      <Button Text="application style overrides" Style="{StaticResource buttonStyle}" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

This results in the appearance shown in the following screenshots:



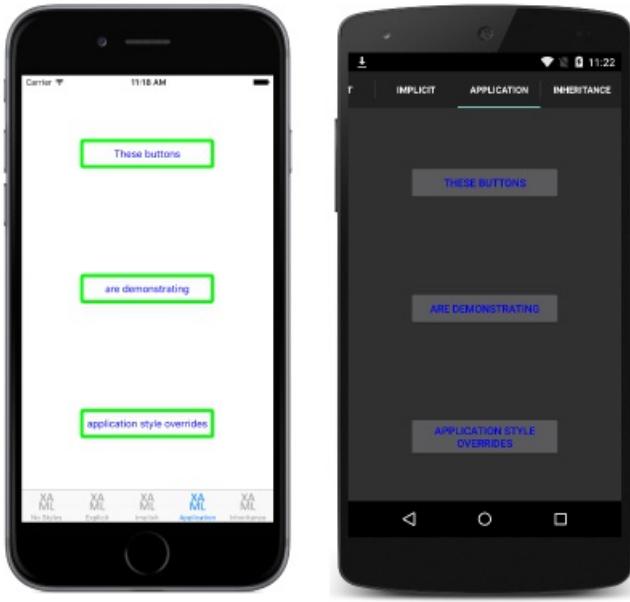
For information about creating styles in a page's [ResourceDictionary](#), see [Explicit Styles](#) and [Implicit Styles](#).

## Overriding Styles

Styles lower in the view hierarchy take precedence over those defined higher up. For example, setting a [Style](#) that sets [Button.TextColor](#) to [Red](#) at the application level will be overridden by a page level style that sets [Button.TextColor](#) to [Green](#). Similarly, a page level style will be overridden by a control level style. In addition, if [Button.TextColor](#) is set directly on a control property, this will take precedence over any styles. This precedence is demonstrated in the following code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ApplicationStylesPage"
    Title="Application" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                ...
                <Setter Property="TextColor" Value="Red" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="buttonStyle" TargetType="Button">
                        ...
                        <Setter Property="TextColor" Value="Blue" />
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            <Button Text="These buttons" Style="{StaticResource buttonStyle}" />
            <Button Text="are demonstrating" Style="{StaticResource buttonStyle}" />
            <Button Text="application style overrides" Style="{StaticResource buttonStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The original [buttonStyle](#), defined at application level, is overridden by the [buttonStyle](#) instance defined at page level. In addition, the page level style is overridden by the control level [buttonStyle](#). Therefore, the [Button](#) instances are displayed with blue text, as shown in the following screenshots:



## Creating a Global Style in C#

`Style` instances can be added to the application's `Resources` collection in C# by creating a new `ResourceDictionary`, and then by adding the `Style` instances to the `ResourceDictionary`, as shown in the following code example:

```
public class App : Application
{
    public App ()
    {
        var buttonStyle = new Style (typeof(Button)) {
            Setters = {
                ...
                new Setter { Property = Button.TextColorProperty, Value = Color.Teal }
            }
        };

        Resources = new ResourceDictionary ();
        Resources.Add ("buttonStyle", buttonStyle);
        ...
    }
    ...
}
```

The constructor defines a single *explicit* style for applying to `Button` instances throughout the application. *Explicit* `Style` instances are added to the `ResourceDictionary` using the `Add` method, specifying a `key` string to refer to the `Style` instance. The `Style` instance can then be applied to any controls of the correct type in the application. However, global styles can be *explicit* or *implicit*.

The following code example shows a C# page applying the `buttonStyle` to the page's `Button` instances:

```
public class ApplicationStylesPageCS : ContentPage
{
    public ApplicationStylesPageCS ()
    {
        ...
        Content = new StackLayout {
            Children = {
                new Button { Text = "These buttons", Style = (Style)Application.Current.Resources
["buttonStyle"] },
                new Button { Text = "are demonstrating", Style = (Style)Application.Current.Resources
["buttonStyle"] },
                new Button { Text = "application styles", Style = (Style)Application.Current.Resources
["buttonStyle"] }
            }
        };
    }
}
```

The `buttonStyle` is applied to the `Button` instances by setting their `Style` properties, and controls the appearance of the `Button` instances.

## Summary

Styles can be made available globally by adding them to the application's `ResourceDictionary`. This helps to avoid duplication of styles across pages or controls.

## Related Links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Style Inheritance in Xamarin.Forms

7/12/2018 • 3 minutes to read • [Edit Online](#)

Styles can inherit from other styles to reduce duplication and enable reuse.

## Style Inheritance in XAML

Style inheritance is performed by setting the `Style.BasedOn` property to an existing `Style`. In XAML, this is achieved by setting the `BasedOn` property to a `StaticResource` markup extension that references a previously created `Style`. In C#, this is achieved by setting the `BasedOn` property to a `Style` instance.

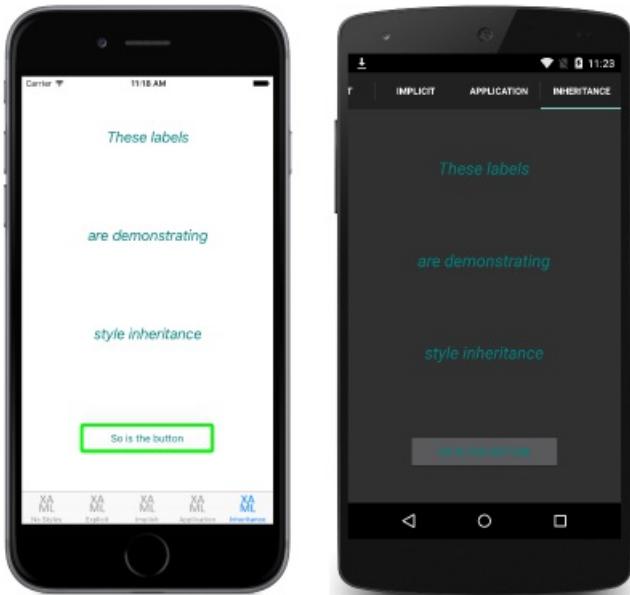
Styles that inherit from a base style can include `Setter` instances for new properties, or use them to override styles from the base style. In addition, styles that inherit from a base style must target the same type, or a type that derives from the type targeted by the base style. For example, if a base style targets `View` instances, styles that are based on the base style can target `View` instances or types that derive from the `View` class, such as `Label` and `Button` instances.

The following code demonstrates *explicit* style inheritance in a XAML page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.StyleInheritancePage"
    Title="Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                <Setter Property="HorizontalOptions"
                    Value="Center" />
                <Setter Property="VerticalOptions"
                    Value="CenterAndExpand" />
            </Style>
            <Style x:Key="labelStyle" TargetType="Label"
                BasedOn="{StaticResource baseStyle}">
                ...
                <Setter Property="TextColor" Value="Teal" />
            </Style>
            <Style x:Key="buttonStyle" TargetType="Button"
                BasedOn="{StaticResource baseStyle}">
                <Setter Property="BorderColor" Value="Lime" />
                ...
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                Style="{StaticResource labelStyle}" />
            ...
            <Button Text="So is the button"
                Style="{StaticResource buttonStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The `baseStyle` targets `View` instances, and sets the `HorizontalOptions` and `VerticalOptions` properties. The `baseStyle` is not set directly on any controls. Instead, `labelStyle` and `buttonStyle` inherit from it, setting additional bindable property values. The `labelStyle` and `buttonStyle` are then applied to the `Label` instances

and `Button` instance, by setting their `Style` properties. This results in the appearance shown in the following screenshots:



#### NOTE

An implicit style can be derived from an explicit style, but an explicit style can't be derived from an implicit style.

### Respecting the Inheritance Chain

A style can only inherit from styles at the same level, or above, in the view hierarchy. This means that:

- An application level resource can only inherit from other application level resources.
- A page level resource can inherit from application level resources, and other page level resources.
- A control level resource can inherit from application level resources, page level resources, and other control level resources.

This inheritance chain is demonstrated in the following code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.StyleInheritancePage"
Title="Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                ...
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="labelStyle" TargetType="Label" BasedOn="{StaticResource baseStyle}">
                        ...
                    </Style>
                    <Style x:Key="buttonStyle" TargetType="Button" BasedOn="{StaticResource baseStyle}">
                        ...
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

In this example, `labelStyle` and `buttonStyle` are control level resources, while `baseStyle` is a page level resource. However, while `labelStyle` and `buttonStyle` inherit from `baseStyle`, it's not possible for `baseStyle` to inherit from `labelStyle` or `buttonStyle`, due to their respective locations in the view hierarchy.

## Style Inheritance in C#

The equivalent C# page, where `Style` instances are assigned directly to the `style` properties of the required controls, is shown in the following code example:

```

public class StyleInheritancePageCS : ContentPage
{
    public StyleInheritancePageCS ()
    {
        var baseStyle = new Style (typeof(View)) {
            Setters = {
                new Setter {
                    Property = View.HorizontalOptionsProperty, Value = LayoutOptions.Center     },
                ...
            }
        };

        var labelStyle = new Style (typeof(Label)) {
            BasedOn = baseStyle,
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Teal     }
            }
        };

        var buttonStyle = new Style (typeof(Button)) {
            BasedOn = baseStyle,
            Setters = {
                new Setter { Property = Button.BorderColorProperty, Value =     Color.Lime },
                ...
            }
        };
        ...

        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels", Style = labelStyle },
                ...
                new Button { Text = "So is the button", Style = buttonStyle }
            }
        };
    }
}

```

The `baseStyle` targets `View` instances, and sets the `HorizontalOptions` and `VerticalOptions` properties. The `baseStyle` is not set directly on any controls. Instead, `labelStyle` and `buttonStyle` inherit from it, setting additional bindable property values. The `labelStyle` and `buttonStyle` are then applied to the `Label` instances and `Button` instance, by setting their `Style` properties.

## Summary

Styles can inherit from other styles to reduce duplication and enable reuse. Style inheritance is performed by setting the `Style.BasedOn` property to an existing `Style`.

## Related Links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Dynamic Styles in Xamarin.Forms

10/15/2018 • 4 minutes to read • [Edit Online](#)

Styles do not respond to property changes, and remain unchanged for the duration of an application. For example, after assigning a Style to a visual element, if one of the Setter instances is modified, removed, or a new Setter instance added, the changes won't be applied to the visual element. However, applications can respond to style changes dynamically at runtime by using dynamic resources.

The `DynamicResource` markup extension is similar to the `StaticResource` markup extension in that both use a dictionary key to fetch a value from a `ResourceDictionary`. However, while the `StaticResource` performs a single dictionary lookup, the `DynamicResource` maintains a link to the dictionary key. Therefore, if the dictionary entry associated with the key is replaced, the change is applied to the visual element. This enables runtime style changes to be made in an application.

The following code example demonstrates *dynamic* styles in a XAML page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DynamicStylesPage" Title="Dynamic"
Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                ...
            </Style>
            <Style x:Key="blueSearchBarStyle"
                TargetType="SearchBar"
                BasedOn="{StaticResource baseStyle}">
                ...
            </Style>
            <Style x:Key="greenSearchBarStyle"
                TargetType="SearchBar">
                ...
            </Style>
            ...
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <SearchBar Placeholder="These SearchBar controls"
                Style="{DynamicResource searchBarStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The `SearchBar` instances use the `DynamicResource` markup extension to reference a `Style` named `searchBarStyle`, which is not defined in the XAML. However, because the `Style` properties of the `SearchBar` instances are set using a `DynamicResource`, the missing dictionary key doesn't result in an exception being thrown.

Instead, in the code-behind file, the constructor creates a `ResourceDictionary` entry with the key `searchBarStyle`, as shown in the following code example:

```

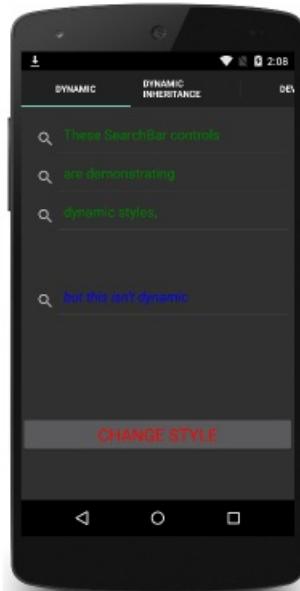
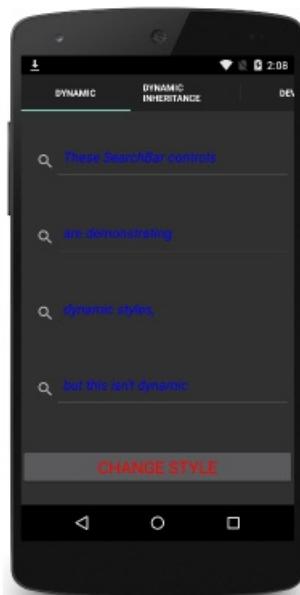
public partial class DynamicStylesPage : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesPage ()
    {
        InitializeComponent ();
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];
    }

    void OnButtonClicked (object sender, EventArgs e)
    {
        if (originalStyle) {
            Resources ["searchBarStyle"] = Resources ["greenSearchBarStyle"];
            originalStyle = false;
        } else {
            Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];
            originalStyle = true;
        }
    }
}

```

When the `OnButtonClicked` event handler is executed, `searchBarStyle` will switch between `blueSearchBarStyle` and `greenSearchBarStyle`. This results in the appearance shown in the following screenshots:



The following code example demonstrates the equivalent page in C#:

```
public class DynamicStylesPageCS : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesPageCS ()
    {
        ...
        var baseStyle = new Style (typeof(View)) {
            ...
        };
        var blueSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var greenSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        ...
        var searchBar1 = new SearchBar { Placeholder = "These SearchBar controls" };
        searchBar1.SetDynamicResource (VisualElement.StyleProperty, "searchBarStyle");
        ...
        Resources = new ResourceDictionary ();
        Resources.Add ("blueSearchBarStyle", blueSearchBarStyle);
        Resources.Add ("greenSearchBarStyle", greenSearchBarStyle);
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];

        Content = new StackLayout {
            Children = { searchBar1, searchBar2, searchBar3, searchBar4, button }
        };
    }
    ...
}
```

In C#, the `earchBar` instances use the `SetDynamicResource` method to reference `searchBarStyle`. The `OnButtonClicked` event handler code is identical to the XAML example, and when executed, `searchBarStyle` will switch between `blueSearchBarStyle` and `greenSearchBarStyle`.

## Dynamic Style Inheritance

Deriving a style from a dynamic style can't be achieved using the `Style.BasedOn` property. Instead, the `Style` class includes the `BaseResourceKey` property, which can be set to a dictionary key whose value might dynamically change.

The following code example demonstrates *dynamic* style inheritance in a XAML page:

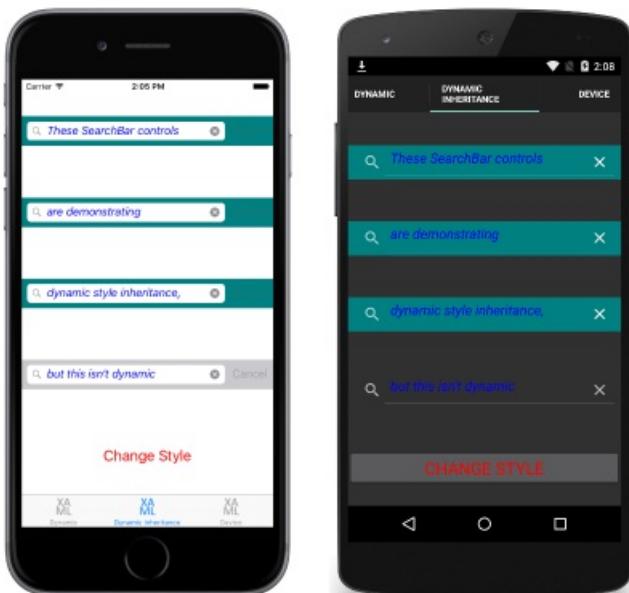
```

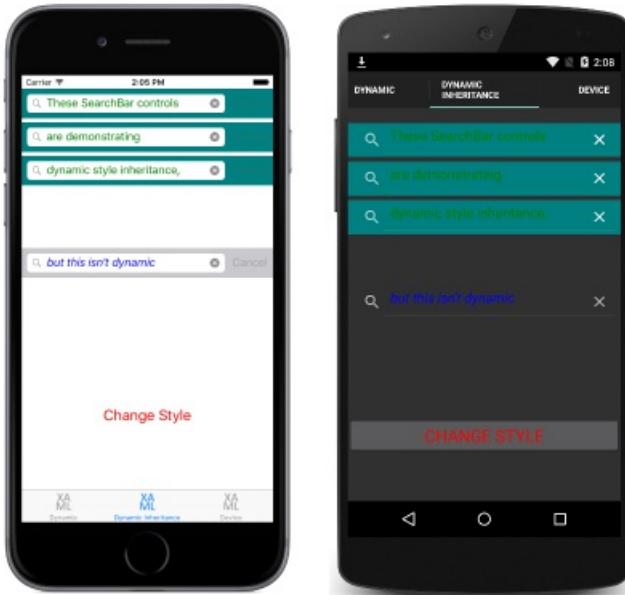
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DynamicStylesInheritancePage"
Title="Dynamic Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                ...
            </Style>
            <Style x:Key="blueSearchBarStyle" TargetType="SearchBar" BasedOn="{StaticResource baseStyle}">
                ...
            </Style>
            <Style x:Key="greenSearchBarStyle" TargetType="SearchBar">
                ...
            </Style>
            <Style x:Key="tealSearchBarStyle" TargetType="SearchBar" BaseResourceKey="searchBarStyle">
                ...
            </Style>
            ...
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <earchBar Text="These SearchBar controls" Style="{StaticResource tealSearchBarStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>

```

The `earchBar` instances use the `StaticResource` markup extension to reference a `Style` named `tealSearchBarStyle`. This `Style` sets some additional properties and uses the `BaseResourceKey` property to reference `searchBarStyle`. The `DynamicResource` markup extension is not required because `tealSearchBarStyle` will not change, except for the `Style` it derives from. Therefore, `tealSearchBarStyle` maintains a link to `searchBarStyle` and is altered when the base style changes.

In the code-behind file, the constructor creates a `ResourceDictionary` entry with the key `searchBarStyle`, as per the previous example that demonstrated dynamic styles. When the `OnButtonClicked` event handler is executed, `searchBarStyle` will switch between `blueSearchBarStyle` and `greenSearchBarStyle`. This results in the appearance shown in the following screenshots:





The following code example demonstrates the equivalent page in C#:

```
public class DynamicStylesInheritancePageCS : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesInheritancePageCS ()
    {
        ...
        var baseStyle = new Style (typeof(View)) {
            ...
        };
        var blueSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var greenSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var tealSearchBarStyle = new Style (typeof(SearchBar)) {
            BaseResourceKey = "searchBarStyle",
            ...
        };
        ...
        Resources = new ResourceDictionary ();
        Resources.Add ("blueSearchBarStyle", blueSearchBarStyle);
        Resources.Add ("greenSearchBarStyle", greenSearchBarStyle);
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];

        Content = new StackLayout {
            Children = {
                new SearchBar { Text = "These SearchBar controls", Style = tealSearchBarStyle },
                ...
            }
        };
    }
}
```

The `tealSearchBarStyle` is assigned directly to the `Style` property of the `earchBar` instances. This `Style` sets some additional properties, and uses the `BaseResourceKey` property to reference `searchBarStyle`. The `SetDynamicResource` method isn't required here because `tealSearchBarStyle` will not change, except for the `Style` it derives from. Therefore, `tealSearchBarStyle` maintains a link to `searchBarStyle` and is altered when the base style changes.

## Summary

Styles do not respond to property changes, and remain unchanged for the duration of an application. However, applications can respond to style changes dynamically at runtime by using dynamic resources. In addition, *dynamic* styles can be derived from with the [BaseResourceKey](#) property.

## Related Links

- [XAML Markup Extensions](#)
- [Dynamic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Device Styles in Xamarin.Forms

7/12/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms includes six dynamic styles, known as *device styles*, in the `Device.Styles` class.

The *device* styles are:

- `BodyStyle`
- `CaptionStyle`
- `ListItemDetailTextStyle`
- `ListItemTextStyle`
- `SubtitleStyle`
- `TitleStyle`

All six styles can only be applied to `Label` instances. For example, a `Label` that's displaying the body of a paragraph might set its `Style` property to `BodyStyle`.

The following code example demonstrates using the *device* styles in a XAML page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DeviceStylesPage" Title="Device"
    Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="myBodyStyle" TargetType="Label"
                BaseResourceKey="BodyStyle">
                <Setter Property="TextColor" Value="Accent" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="Title style"
                Style="{DynamicResource TitleStyle}" />
            <Label Text="Subtitle text style"
                Style="{DynamicResource SubtitleStyle}" />
            <Label Text="Body style"
                Style="{DynamicResource BodyStyle}" />
            <Label Text="Caption style"
                Style="{DynamicResource CaptionStyle}" />
            <Label Text="List item detail text style"
                Style="{DynamicResource ListItemDetailTextStyle}" />
            <Label Text="List item text style"
                Style="{DynamicResource ListItemTextStyle}" />
            <Label Text="No style" />
            <Label Text="My body style"
                Style="{StaticResource myBodyStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The device styles are bound to using the `DynamicResource` markup extension. The dynamic nature of the styles can be seen in iOS by changing the **Accessibility** settings for text size. The appearance of the *device* styles is different on each platform, as shown in the following screenshots:

Title style

Subtitle style

Body style

Caption style

List item detail text style

List item text style

No style

My body style

Title style

Subtitle style

Body style

Caption style

List item detail text style

List item text style

No style

My body style

Title style

Subtitle style

Body style

Caption style

List item detail text style

List item text style

No style

My body style

iOS

Android

Windows Phone

Device styles can also be derived from by setting the `BaseResourceKey` property to the key name for the device style. In the code example above, `myBodyStyle` inherits from `BodyStyle` and sets an accented text color. For more information about dynamic style inheritance, see [Dynamic Style Inheritance](#).

The following code example demonstrates the equivalent page in C#:

```
public class DeviceStylesPageCS : ContentPage
{
    public DeviceStylesPageCS ()
    {
        var myBodyStyle = new Style (typeof(Label)) {
            BaseResourceKey = Device.Styles.BodyStyleKey,
            Setters = {
                new Setter {
                    Property = Label.TextColorProperty,
                    Value = Color.Accent
                }
            }
        };

        Title = "Device";
        Icon = "csharp.png";
        Padding = new Thickness (0, 20, 0, 0);

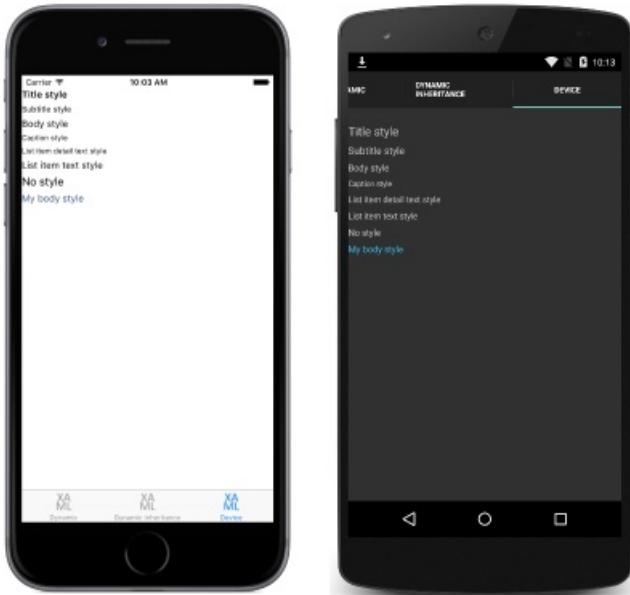
        Content = new StackLayout {
            Children = {
                new Label { Text = "Title style", Style = Device.Styles.TitleStyle },
                new Label { Text = "Subtitle style", Style = Device.Styles.SubtitleStyle },
                new Label { Text = "Body style", Style = Device.Styles.BodyStyle },
                new Label { Text = "Caption style", Style = Device.Styles.CaptionStyle },
                new Label { Text = "List item detail text style",
                    Style = Device.Styles.ListItemDetailTextStyle },
                new Label { Text = "List item text style", Style = Device.Styles.ListItemTextStyle },
                new Label { Text = "No style" },
                new Label { Text = "My body style", Style = myBodyStyle }
            }
        };
    }
}
```

The `Style` property of each `Label` instance is set to the appropriate property from the `Devices.Styles` class.

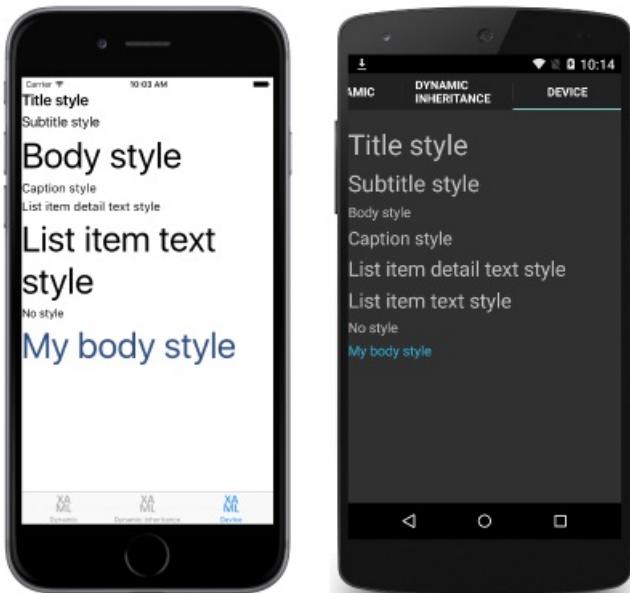
## Accessibility

The *device* styles respect accessibility preferences, so font sizes will change as the accessibility preferences are altered on each platform. Therefore, to support accessible text, ensure that the *device* styles are used as the basis for any text styles within your application.

The following screenshots demonstrate the device styles on each platform, with the smallest accessible font size:



The following screenshots demonstrate the device styles on each platform, with the largest accessible font size:



## Summary

Xamarin.Forms includes six *dynamic* styles, known as *device* styles, in the [Devices.Styles](#) class. All six styles can only be applied to [Label](#) instances.

## Related Links

- [Text Styles](#)
- [XAML Markup Extensions](#)
- [Dynamic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [Device.Styles](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

# Styling Xamarin.Forms apps using Cascading Style Sheets (CSS)

11/20/2018 • 12 minutes to read • [Edit Online](#)

*Xamarin.Forms supports styling visual elements using Cascading Style Sheets (CSS).*

Xamarin.Forms applications can be styled using CSS. A style sheet consists of a list of rules, with each rule consisting of one or more selectors and a declaration block. A declaration block consists of a list of declarations in braces, with each declaration consisting of a property, a colon, and a value. When there are multiple declarations in a block, a semi-colon is inserted as a separator. The following code example shows some Xamarin.Forms compliant CSS:

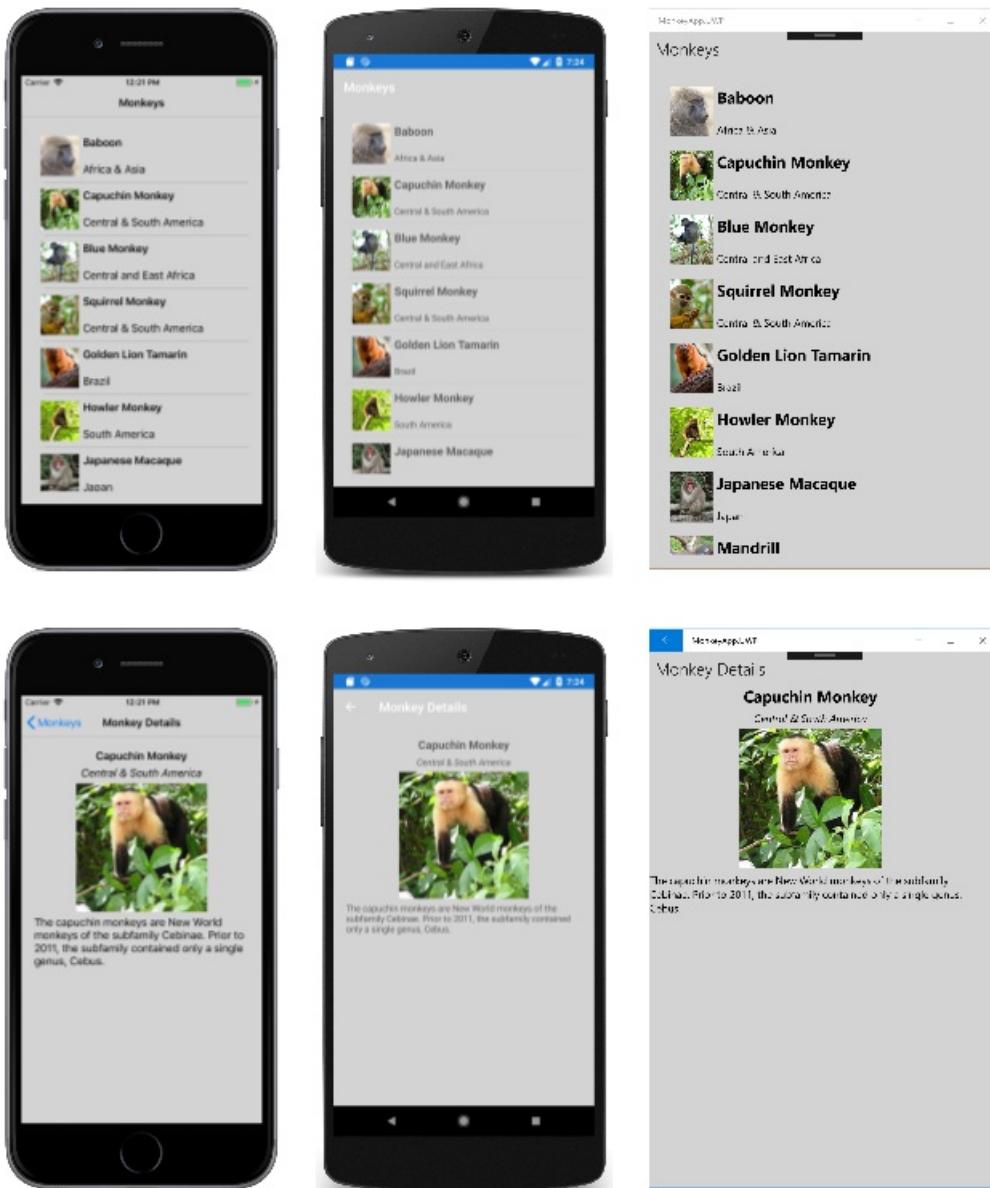
```
navigationpage {  
    -xf-bar-background-color: lightgray;  
}  
  
^contentpage {  
    background-color: lightgray;  
}  
  
.listView {  
    background-color: lightgray;  
}  
  
.stacklayout {  
    margin: 20;  
}  
  
.mainPageTitle {  
    font-style: bold;  
    font-size: medium;  
}  
  
.mainPageSubtitle {  
    margin-top: 15;  
}  
  
.detailPageTitle {  
    font-style: bold;  
    font-size: medium;  
    text-align: center;  
}  
  
.detailPageSubtitle {  
    text-align: center;  
    font-style: italic;  
}  
  
.listview image {  
    height: 60;  
    width: 60;  
}  
  
.stacklayout>image {  
    height: 200;  
    width: 200;  
}
```

In Xamarin.Forms, CSS style sheets are parsed and evaluated at runtime, rather than compile time, and style sheets are re-parsed on use.

#### NOTE

Currently, all of the styling that's possible with XAML styling cannot be performed with CSS. However, XAML styles can be used to supplement CSS for properties that are currently unsupported by Xamarin.Forms. For more information about XAML styles, see [Styling Xamarin.Forms Apps using XAML Styles](#).

The [MonkeyAppCSS](#) sample demonstrates using CSS to style a simple app, and is shown in the following screenshots:



## Consuming a style sheet

The process for adding a style sheet to a solution is as follows:

1. Add an empty CSS file to your .NET Standard library project.
2. Set the build action of the CSS file to **EmbeddedResource**.

### Loading a style sheet

There are a number of approaches that can be used to load a style sheet.

## XAML

A style sheet can be loaded and parsed with the `StyleSheet` class before being added to a `ResourceDictionary`:

```
<Application ...>
<Application.Resources>
    <StyleSheet Source="/Assets/styles.css" />
</Application.Resources>
</Application>
```

The `StyleSheet.Source` property specifies the style sheet as a URI relative to the location of the enclosing XAML file, or relative to the project root if the URI starts with a `/`.

### WARNING

The CSS file will fail to load if its build action is not set to **EmbeddedResource**.

Alternatively, a style sheet can be loaded and parsed with the `StyleSheet` class, before being added to a `ResourceDictionary`, by inlining it in a `CDATA` section:

```
<ContentPage ...>
<ContentPage.Resources>
    <StyleSheet>
        <![CDATA[
            ^contentpage {
                background-color: lightgray;
            }
        ]]>
    </StyleSheet>
</ContentPage.Resources>
...
</ContentPage>
```

For more information about resource dictionaries, see [Resource Dictionaries](#).

## C#

In C#, a style sheet can be loaded as an embedded resource and added to a `ResourceDictionary`:

```
public partial class MyPage : ContentPage
{
    public MyPage()
    {
        InitializeComponent();

        this.Resources.Add(StyleSheet.FromAssemblyResource(
            IntrospectionExtensions.GetTypeInfo(typeof(MyPage)).Assembly,
            "MyProject.Assets.styles.css"));
    }
}
```

The first argument to the `StyleSheet.FromAssemblyResource` method is the assembly containing the style sheet, while the second argument is a `string` representing the resource identifier. The resource identifier can be obtained from the **Properties** window when the CSS file is selected.

Alternatively, a style sheet can be loaded from a `StringReader` and added to a `ResourceDictionary`:

```
public partial class MyPage : ContentPage
{
    public MyPage()
    {
        InitializeComponent();

        using (var reader = new StringReader("contentpage { background-color: lightgray; }"))
        {
            this.Resources.Add(StyleSheet.FromReader(reader));
        }
    }
}
```

The argument to the `StyleSheet.FromReader` method is the `TextReader` that has read the style sheet.

## Selecting elements and applying properties

CSS uses selectors to determine which elements to target. Styles with matching selectors are applied consecutively, in definition order. Styles defined on a specific item are always applied last. For more information about supported selectors, see [Selector Reference](#).

CSS uses properties to style a selected element. Each property has a set of possible values, and some properties can affect any type of element, while others apply to groups of elements. For more information about supported properties, see [Property Reference](#).

### Selecting elements by type

Elements in the visual tree can be selected by type with the case insensitive `element` selector:

```
stacklayout {
    margin: 20;
}
```

This selector identifies any `StackLayout` elements on pages that consume the style sheet, and sets their margins to a uniform thickness of 20.

#### NOTE

The `element` selector does not identify sub-classes of the specified type.

### Selecting elements by base class

Elements in the visual tree can be selected by base class with the case insensitive `^base` selector:

```
^contentpage {
    background-color: lightgray;
}
```

This selector identifies any `ContentPage` elements that consume the style sheet, and sets their background color to `lightgray`.

#### NOTE

The `^base` selector is specific to Xamarin.Forms, and isn't part of the CSS specification.

### Selecting an element by name

Individual elements in the visual tree can be selected with the case sensitive `#id` selector:

```
#listView {  
    background-color: lightgray;  
}
```

This selector identifies the element whose `StyleId` property is set to `listView`. However, if the `StyleId` property is not set, the selector will fall back to using the `x:Name` of the element. Therefore, in the following XAML example, the `#listView` selector will identify the `ListView` whose `x:Name` attribute is set to `listView`, and will set its background color to `lightgray`.

```
<ContentPage ...>  
    <ContentPage.Resources>  
        <StyleSheet Source="/Assets/styles.css" />  
    </ContentPage.Resources>  
    <StackLayout>  
        <ListView x:Name="listView" ...>  
        ...  
    </ListView>  
    </StackLayout>  
</ContentPage>
```

## Selecting elements with a specific class attribute

Elements with a specific class attribute can be selected with the case sensitive `.class` selector:

```
.detailPageTitle {  
    font-style: bold;  
    font-size: medium;  
    text-align: center;  
}  
  
.detailPageSubtitle {  
    text-align: center;  
    font-style: italic;  
}
```

A CSS class can be assigned to a XAML element by setting the `StyleClass` property of the element to the CSS class name. Therefore, in the following XAML example, the styles defined by the `.detailPageTitle` class are assigned to the first `Label`, while the styles defined by the `.detailPageSubtitle` class are assigned to the second `Label`.

```
<ContentPage ...>  
    <ContentPage.Resources>  
        <StyleSheet Source="/Assets/styles.css" />  
    </ContentPage.Resources>  
    <ScrollView>  
        <StackLayout>  
            <Label ... StyleClass="detailPageTitle" />  
            <Label ... StyleClass="detailPageSubtitle"/>  
            ...  
        </StackLayout>  
    </ScrollView>  
</ContentPage>
```

## Selecting child elements

Child elements in the visual tree can be selected with the case insensitive `element element` selector:

```
listview image {  
    height: 60;  
    width: 60;  
}
```

This selector identifies any `Image` elements that are children of `ListView` elements, and sets their height and width to 60. Therefore, in the following XAML example, the `listview image` selector will identify the `Image` that's a child of the `ListView`, and sets its height and width to 60.

```
<ContentPage ...>  
    <ContentPage.Resources>  
        <StyleSheet Source="/Assets/styles.css" />  
    </ContentPage.Resources>  
    <StackLayout>  
        <ListView ...>  
            <ListView.ItemTemplate>  
                <DataTemplate>  
                    <ViewCell>  
                        <Grid>  
                            ...  
                            <Image ... />  
                            ...  
                        </Grid>  
                    </ViewCell>  
                </DataTemplate>  
            </ListView.ItemTemplate>  
        </ListView>  
    </StackLayout>  
</ContentPage>
```

#### NOTE

The `element element` selector does not require the child element to be a *direct* child of the parent – the child element may have a different parent. Selection occurs provided that an ancestor is the specified first element.

### Selecting direct child elements

Direct child elements in the visual tree can be selected with the case insensitive `element>element` selector:

```
stacklayout>image {  
    height: 200;  
    width: 200;  
}
```

This selector identifies any `Image` elements that are direct children of `StackLayout` elements, and sets their height and width to 200. Therefore, in the following XAML example, the `stacklayout>image` selector will identify the `Image` that's a direct child of the `StackLayout`, and sets its height and width to 200.

```

<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </ContentPage.Resources>
    <ScrollView>
        <StackLayout>
            ...
            <Image ... />
            ...
        </StackLayout>
    </ScrollView>
</ContentPage>

```

#### NOTE

The `element>element` selector requires that the child element is a *direct* child of the parent.

## Selector reference

The following CSS selectors are supported by Xamarin.Forms:

SELECTOR	EXAMPLE	DESCRIPTION
<code>.class</code>	<code>.header</code>	Selects all elements with the <code>StyleClass</code> property containing 'header'. Note that this selector is case sensitive.
<code>#id</code>	<code>#email</code>	Selects all elements with <code>StyleId</code> set to <code>email</code> . If <code>StyleId</code> is not set, fallback to <code>x:Name</code> . When using XAML, <code>x:Name</code> is preferred over <code>StyleId</code> . Note that this selector is case sensitive.
<code>*</code>	<code>*</code>	Selects all elements.
<code>element</code>	<code>label</code>	Selects all elements of type <code>Label</code> , but not sub-classes. Note that this selector is case insensitive.
<code>^base</code>	<code>^contentpage</code>	Selects all elements with <code>ContentPage</code> as the base class, including <code>ContentPage</code> itself. Note that this selector is case insensitive and isn't part of the CSS specification.
<code>element,element</code>	<code>label,button</code>	Selects all <code>Button</code> elements and all <code>Label</code> elements. Note that this selector is case insensitive.
<code>element element</code>	<code>stacklayout label</code>	Selects all <code>Label</code> elements inside a <code>StackLayout</code> . Note that this selector is case insensitive.

SELECTOR	EXAMPLE	DESCRIPTION
<code>element&gt;element</code>	<code>stacklayout&gt;label</code>	Selects all <code>Label</code> elements with <code>StackLayout</code> as a direct parent. Note that this selector is case insensitive.
<code>element+element</code>	<code>label+entry</code>	Selects all <code>Entry</code> elements directly after a <code>Label</code> . Note that this selector is case insensitive.
<code>element~element</code>	<code>label~entry</code>	Selects all <code>Entry</code> elements preceded by a <code>Label</code> . Note that this selector is case insensitive.

Styles with matching selectors are applied consecutively, in definition order. Styles defined on a specific item are always applied last.

#### TIP

Selectors can be combined without limitation, such as `StackLayout>ContentView>label.email`.

The following selectors are currently unsupported:

- `[attribute]`
- `@media` and `@supports`
- `:` and `::`

#### NOTE

Specificity, and specificity overrides are unsupported.

## Property reference

The following CSS properties are supported by Xamarin.Forms (in the **Values** column, types are *italic*, while string literals are `gray`):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
<code>align-content</code>	<code>FlexLayout</code>	<code>stretch</code>   <code>center</code>   <code>start</code>   <code>end</code>   <code>space-between</code>   <code>space-around</code>   <code>space-evenly</code>   <code>flex-start</code>   <code>flex-end</code>   <code>space-between</code>   <code>space-around</code>   <code>initial</code>	<code>align-content: space-between;</code>
<code>align-items</code>	<code>FlexLayout</code>	<code>stretch</code>   <code>center</code>   <code>start</code>   <code>end</code>   <code>flex-start</code>   <code>flex-end</code>   <code>initial</code>	<code>align-items: flex-start;</code>

PROPERTY	APPLIES TO	VALUES	EXAMPLE
align-self	VisualElement	auto   stretch   center   start   end   flex-start   flex-end   initial	align-self: flex-end;
background-color	VisualElement	color   initial	background-color: springgreen;
background-image	Page	string   initial	background-image: bg.png;
border-color	Button , Frame , ImageButton	color   initial	border-color: #9acd32;
border-radius	BoxView	double   initial	border-radius: 10;
border-width	Button	double   initial	border-width: .5;
color	ActivityIndicator , BoxView , Button , DatePicker , Editor , Entry , Label , Picker , ProgressBar , SearchBar , Switch , TimePicker	color   initial	color: rgba(255, 0, 0, 0.3);
column-gap	Grid	double   initial	column-gap: 9;
direction	VisualElement	ltr   rtl   inherit   initial	direction: rtl;
flex-direction	FlexLayout	column   columnreverse   row   rowreverse   row-reverse   column-reverse   initial	flex-direction: column-reverse;
flex-basis	VisualElement	float   auto   initial . In addition, a percentage in the range 0% to 100% can be specified with the % sign.	flex-basis: 25%;
flex-grow	VisualElement	float   initial	flex-grow: 1.5;
flex-shrink	VisualElement	float   initial	flex-shrink: 1;
flex-wrap	VisualElement	nowrap   wrap   reverse   wrap-reverse   initial	flex-wrap: wrap-reverse;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
font-family	Button , DatePicker , Editor , Entry , Label , Picker , SearchBar , TimePicker , Span	string   initial	font-family: Consolas;
font-size	Button , DatePicker , Editor , Entry , Label , Picker , SearchBar , TimePicker , Span	double   namedsize   initial	font-size: 12;
font-style	Button , DatePicker , Editor , Entry , Label , Picker , SearchBar , TimePicker , Span	bold   italic   initial	font-style: bold;
height	VisualElement	double   initial	min-height: 250;
justify-content	FlexLayout	start   center   end   spacebetween   spacearound   spaceevenly   flex-start   flex-end   space-between   space-around   initial	justify-content: flex-end;
line-height	Label , Span	double   initial	line-height: 1.8;
margin	View	thickness   initial	margin: 6 12;
margin-left	View	thickness   initial	margin-left: 3;
margin-top	View	thickness   initial	margin-top: 2;
margin-right	View	thickness   initial	margin-right: 1;
margin-bottom	View	thickness   initial	margin-bottom: 6;
max-lines	Label	int   initial	max-lines: 2;
min-height	VisualElement	double   initial	min-height: 50;
min-width	VisualElement	double   initial	min-width: 112;
opacity	VisualElement	double   initial	opacity: .3;
order	VisualElement	int   initial	order: -1;
padding	Button , ImageButton , Layout , Page	thickness   initial	padding: 6 12 12;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
padding-left	Button , ImageButton , Layout , Page	double   initial	padding-left: 3;
padding-top	Button , ImageButton , Layout , Page	double   initial	padding-top: 4;
padding-right	Button , ImageButton , Layout , Page	double   initial	padding-right: 2;
padding-bottom	Button , ImageButton , Layout , Page	double   initial	padding-bottom: 6;
position	FlexLayout	relative   absolute   initial	position: absolute;
row-gap	Grid	double   initial	row-gap: 12;
text-align	Entry , EntryCell , Label , SearchBar	left   top   right   bottom   start   center   middle   end   initial . left and right should be avoided in right-to-left environments.	text-align: right;
text-decoration	Label , Span	none   underline   strikethrough   line-through   initial	text-decoration: underline, line-through;
transform	VisualElement	none , rotate , rotateX , rotateY , scale , scaleX , scaleY , translate , translateX , translateY , initial	transform: rotate(180), scaleX(2.5);
transform-origin	VisualElement	double, double   initial	transform-origin: 7.5, 12.5;
vertical-align	Label	left   top   right   bottom   start   center   middle   end   initial	vertical-align: bottom;
visibility	VisualElement	true   visible   false   hidden   collapse   initial	visibility: hidden;
width	VisualElement	double   initial	min-width: 320;

The following Xamarin.Forms specific CSS properties are also supported (in the **Values** column, types are *italic*, while string literals are gray):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-xf-placeholder	Entry , Editor , SearchBar	quoted text   initial	-xf-placeholder: Enter name;
-xf-placeholder-color	Entry , Editor , SearchBar	color   initial	-xf-placeholder-color: green;
-xf-max-length	Entry , Editor	int   initial	-xf-max-length: 20;
-xf-bar-background-color	NavigationPage , TabbedPage	color   initial	-xf-bar-background-color: teal;
-xf-bar-text-color	NavigationPage , TabbedPage	color   initial	-xf-bar-text-color: gray
-xf-orientation	ScrollView , StackLayout	horizontal   vertical   both   initial . both is only supported on a ScrollView .	-xf-orientation: horizontal;
-xf-horizontal-scroll-bar-visibility	ScrollView	default   always   never   initial	-xf-horizontal-scroll-bar-visibility: never;
-xf-vertical-scroll-bar-visibility	ScrollView	default   always   never   initial	-xf-vertical-scroll-bar-visibility: always;
-xf-min-track-color	Slider	color   initial	-xf-min-track-color: yellow;
-xf-max-track-color	Slider	color   initial	-xf-max-track-color: red;
-xf-thumb-color	Slider	color   initial	-xf-thumb-color: limegreen;
-xf-spacing	StackLayout	double   initial	-xf-spacing: 8;

#### NOTE

initial is a valid value for all properties. It clears the value (resets to default) that was set from another style.

The following properties are currently unsupported:

- all: initial .
- Layout properties (box, or grid).
- Shorthand properties, such as font , and border .

In addition, there's no inherit value and so inheritance isn't supported. Therefore you can't, for example, set the font-size property on a layout and expect all the Label instances in the layout to inherit the value. The one exception is the direction property, which has a default value of inherit .

#### Color

The following `color` values are supported:

- `X11 colors`, which match CSS colors, UWP pre-defined colors, and Xamarin.Forms colors. Note that these color values are case insensitive.
- hex colors: `#rgb`, `#argb`, `#rrggbb`, `#aarrggbb`
- `rgb` colors: `rgb(255,0,0)`, `rgb(100%,0%,0%)`. Values are in the range 0-255, or 0%-100%.
- `rgba` colors: `rgba(255, 0, 0, 0.8)`, `rgba(100%, 0%, 0%, 0.8)`. The opacity value is in the range 0.0-1.0.
- `hsl` colors: `hsl(120, 100%, 50%)`. The h value is in the range 0-360, while s and l are in the range 0%-100%.
- `hsla` colors: `hsla(120, 100%, 50%, .8)`. The opacity value is in the range 0.0-1.0.

## Thickness

One, two, three, or four `thickness` values are supported, each separated by white space:

- A single value indicates uniform thickness.
- Two values indicate vertical then horizontal thickness.
- Three values indicate top, then horizontal (left and right), then bottom thickness.
- Four values indicate top, then right, then bottom, then left thickness.

### NOTE

CSS `thickness` values differ from XAML `Thickness` values. For example, in XAML a two-value `Thickness` indicates horizontal then vertical thickness, while a four-value `Thickness` indicates left, then top, then right, then bottom thickness. In addition, XAML `Thickness` values are comma delimited.

## NamedSize

The following case insensitive `namedsize` values are supported:

- `default`
- `micro`
- `small`
- `medium`
- `large`

The exact meaning of each `namedsize` value is platform-dependent and view-dependent.

## CSS in Xamarin.Forms with Xamarin.University

### Xamarin.Forms 3.0 CSS, by [Xamarin University](#)

## Related Links

- [MonkeyAppCSS \(sample\)](#)
- [Resource Dictionaries](#)
- [Styling Xamarin.Forms Apps using XAML Styles](#)

# Xamarin.Forms TableView

10/18/2018 • 6 minutes to read • [Edit Online](#)

[TableView](#) is a view for displaying scrollable lists of data or choices where there are rows that don't share the same template. Unlike [ListView](#), TableView does not have the concept of an `ItemsSource`, so items must be added as children manually.

This guide is composed of the following sections:

- [Use Cases](#) – when to use TableView instead of ListView or a custom view.
- [TableView Structure](#) – the hierarchy of views that is needed within a TableView.
- [TableView Appearance](#) – the customization options for TableView.
- [Built-In Cells](#) – built-in cell options, including `EntryCell` and `SwitchCell`.
- [Custom Cells](#) – how to make your own custom cells.



## Use Cases

TableView is useful when:

- presenting a list of settings,
- collecting data in a form, or
- showing data that is presented differently from row to row (e.g. numbers, percentages and images).

TableView handles scrolling and laying out rows in attractive sections, a common need for the above scenarios. The `TableView` control uses each platform's underlying equivalent view when available, creating a native look for each platform.

## TableView Structure

Elements in a `TableView` are organized into sections. At the root of the `TableView` is the `TableRoot`, which is parent to one or more `TableSections`:

```

Content = new TableView {
    Root = new TableRoot {
        new TableSection...
    },
    Intent = TableIntent.Settings
};

```

Each `TableSection` consists of a heading and one or more `ViewCells`. Here we see the `TableSection`'s `Title` property set to "Ring" in the constructor:

```

var section = new TableSection ("Ring") { //TableSection constructor takes title as an optional parameter
    new SwitchCell {Text = "New Voice Mail"},
    new SwitchCell {Text = "New Mail", On = true}
};

```

To accomplish the same layout as above in XAML:

```

<TableView Intent="Settings">
    <TableRoot>
        <TableSection Title="Ring">
            <SwitchCell Text="New Voice Mail" />
            <SwitchCell Text="New Mail" On="true" />
        </TableSection>
    </TableRoot>
</TableView>

```

## TableView Appearance

`TableView` exposes the `Intent` property, which is an enumeration of the following options:

- **Data** – for use when displaying data entries. Note that [ListView](#) may be a better option for scrolling lists of data.
- **Form** – for use when the `TableView` is acting as a Form.
- **Menu** – for use when presenting a menu of selections.
- **Settings** – for use when displaying a list of configuration settings.

The `TableIntent` you choose may impact how the `TableView` appears on each platform. Even if there are not clear differences, it is a best practice to select the `TableIntent` that most closely matches how you intend to use the table.

## Built-In Cells

Xamarin.Forms comes with built-in cells for collecting and displaying information. Although `ListView` and `TableView` can use all of the same cells, the following are the most relevant for a `TableView` scenario:

- **SwitchCell** – for presenting and capturing a true/false state, along with a text label.
- **EntryCell** – for presenting and capturing text.

See [ListView Cell Appearance](#) for a detailed description of `TextCell` and `ImageCell`.

### SwitchCell

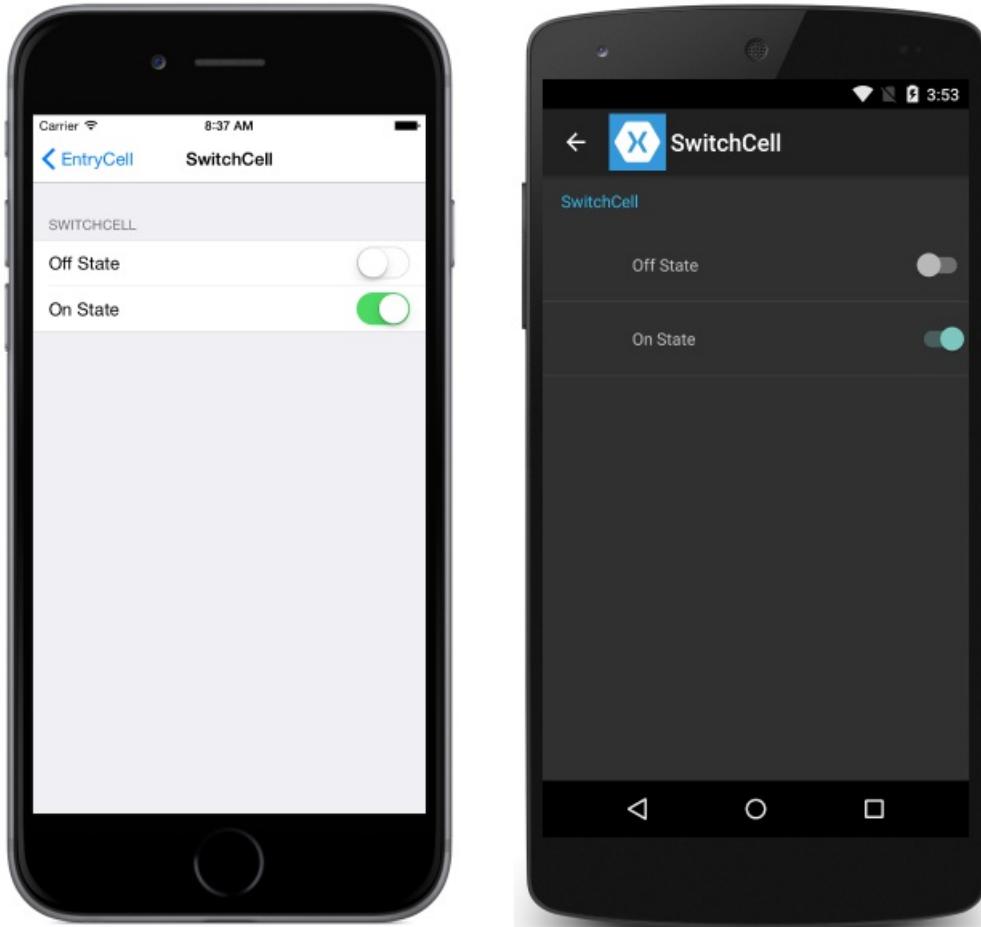
`SwitchCell` is the control to use for presenting and capturing an on/off or `true` / `false` state.

SwitchCells have one line of text to edit and an on/off property. Both of these properties are bindable.

- `Text` – text to display beside the switch.

- `On` – whether the switch is displayed as on or off.

Note that the `SwitchCell` exposes the `OnChanged` event, allowing you to respond to changes in the cell's state.

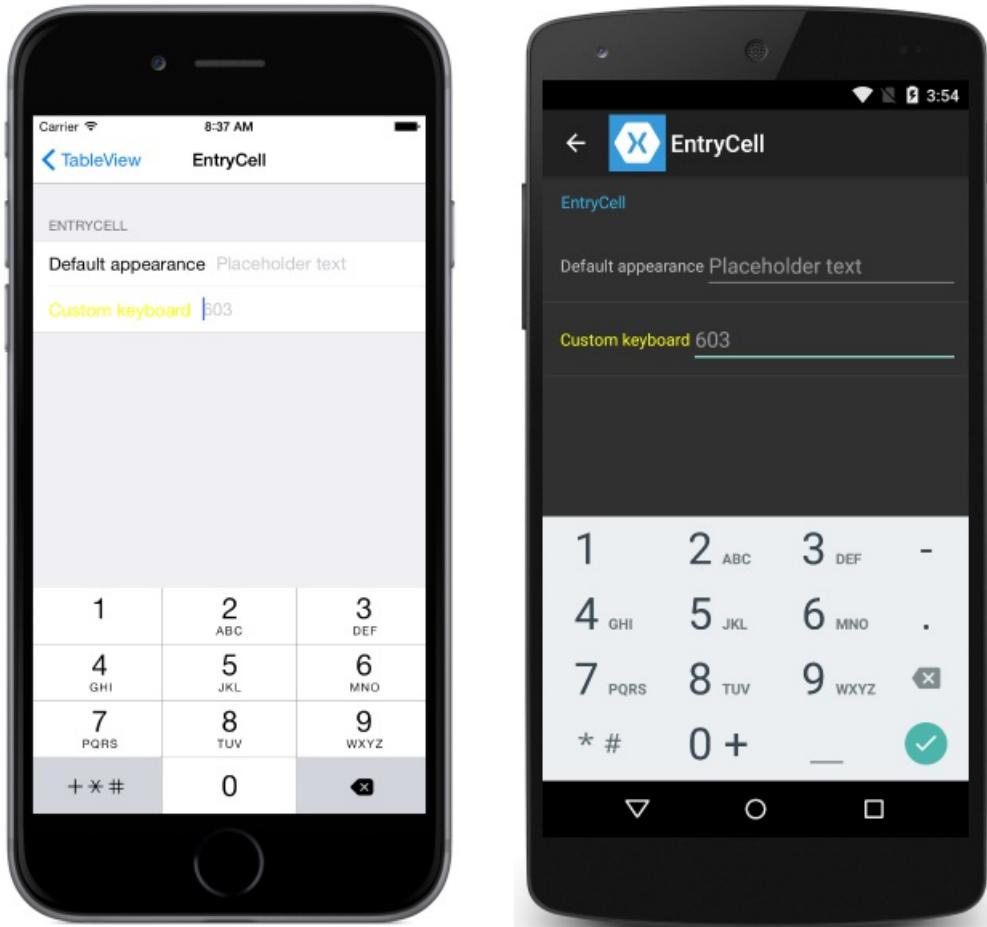


## EntryCell

`EntryCell` is useful when you need to display text data that the user can edit. `EntryCell`s offer the following properties that can be customized:

- `Keyboard` – The keyboard to display while editing. There are options for things like numeric values, email, phone numbers, etc. [See the API docs.](#)
- `Label` – The label text to display to the right of the text entry field.
- `LabelColor` – The color of the label text.
- `Placeholder` – Text to display in the entry field when it is null or empty. This text disappears when text entry begins.
- `Text` – The text in the entry field.
- `HorizontalTextAlignment` – The horizontal alignment of the text. Can be center, left, or right aligned. [See the API docs.](#)

Note that `EntryCell` exposes the `Completed` event, which is fired when the user hits 'done' on the keyboard while editing text.

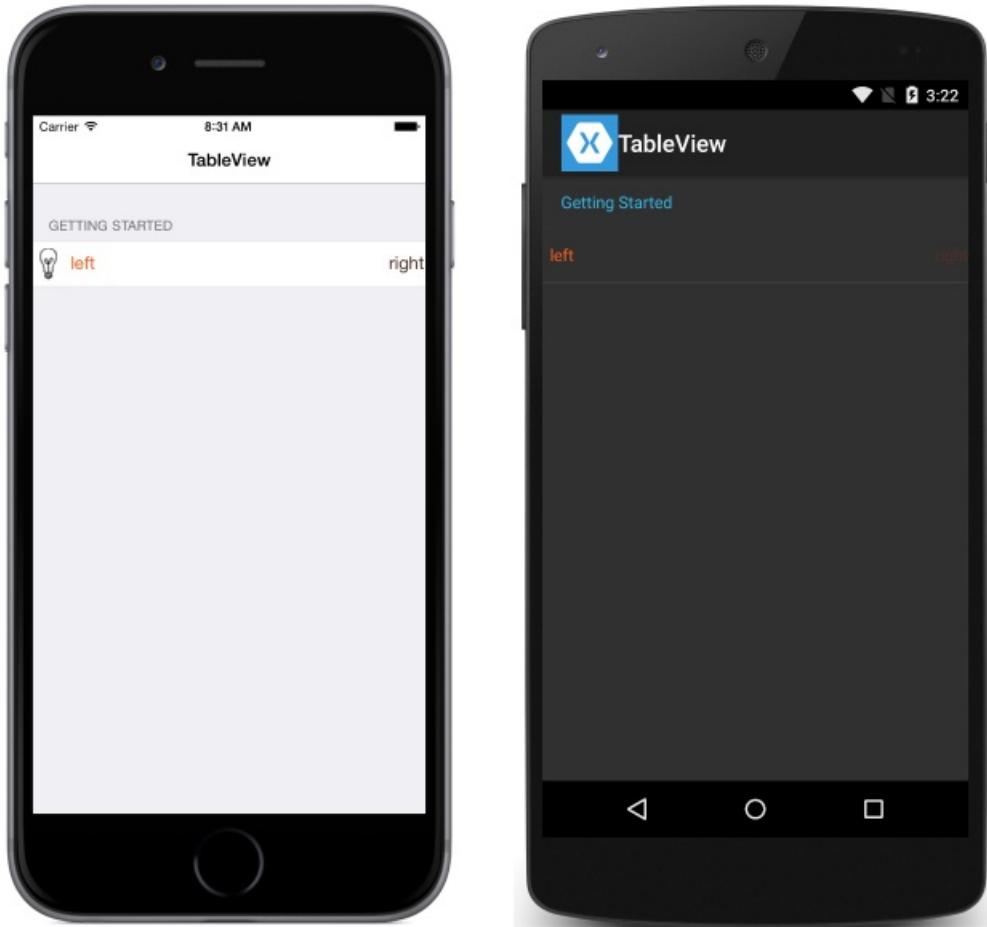


## Custom Cells

When the built-in cells aren't enough, custom cells can be used to present and capture data in the way that makes sense for your app. For example, you may want to present a slider to allow a user to choose the opacity of an image.

All custom cells must derive from [ViewCell](#), the same base class that all of the built-in cell types use.

This is an example of a custom cell:



## XAML

The XAML to create the above layout is below:

```
<?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="DemoTableView.TablePage" Title="TableView">
    <ContentPage.Content>
        <TableView Intent="Settings">
            <TableRoot>
                <TableSection Title="Getting Started">
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <Image Source="bulb.png" />
                            <Label Text="left"
                                TextColor="#f35e20" />
                            <Label Text="right"
                                HorizontalOptions="EndAndExpand"
                                TextColor="#503026" />
                        </StackLayout>
                    </ViewCell>
                </TableSection>
            </TableRoot>
        </TableView>
    </ContentPage.Content>
</ContentPage>
```

The XAML above is doing a lot. Let's break it down:

- The root element under the `TableView` is the `TableRoot`.
- There is a `TableSection` immediately underneath the `TableRoot`.
- The `ViewCell` is defined directly under the `TableSection`. Unlike `ListView`, `TableView` does not require that

custom (or any) cells are defined in an `ItemTemplate`.

- A StackLayout is used to manage the layout of the custom cell. Any layout could be used here.

## C#

Because `TableView` works with static data, or data that is manually changed, it does not have the concept of an item template. Instead, custom cells can be manually created and put into the table. Note that the technique of creating a custom cell that inherits from `ViewCell`, then adding it to the `TableView` like you would a built-in cell, is also supported. Here is the c# code to accomplish the layout above:

```
var table = new TableView();
table.Intent = TableIntent.Settings;
var layout = new StackLayout() { Orientation = StackOrientation.Horizontal };
layout.Children.Add (new Image() {Source = "bulb.png"});
layout.Children.Add (new Label() {
    Text = "left",
    TextColor = Color.FromHex("#f35e20"),
    VerticalOptions = LayoutOptions.Center
});
layout.Children.Add (new Label () {
    Text = "right",
    TextColor = Color.FromHex ("#503026"),
    VerticalOptions = LayoutOptions.Center,
    HorizontalOptions = LayoutOptions.EndAndExpand
});
table.Root = new TableRoot () {
    new TableSection("Getting Started") {
        new ViewCell() {View = layout}
    }
};
Content = table;
```

The C# above is doing a lot. Let's break it down:

- The root element under the `TableView` is the `TableRoot`.
- There is a `TableSection` immediately underneath the `TableRoot`.
- The `ViewCell` is defined directly under the `TableSection`. Unlike `ListView`, `TableView` does not require that custom (or any) cells are defined in an `ItemTemplate`.
- A `StackLayout` is used to manage the layout of the custom cell. Any layout could be used here.

Note that a class for the custom cell is never defined. Instead, the `ViewCell`'s `view` property is set for a particular instance of `ViewCell`.

## Row Height

The `TableView` class has two properties that can be used to change the row height of cells:

- `RowHeight` – sets the height of each row to an `int`.
- `HasUnevenRows` – rows have varying heights if set to `true`. Note that when setting this property to `true`, row heights will automatically be calculated and applied by Xamarin.Forms.

When the height of content in a cell in a `TableView` is changed, the row height is implicitly updated on Android and the Universal Windows Platform (UWP). However, on iOS it must be forced to update by setting the `HasUnevenRows` property to `true` and by calling the `Cell.ForceUpdateSize` method.

The following XAML example shows a `TableView` that contains a `ViewCell`:

```

<ContentPage ...>
    <TableView ...
        HasUnevenRows="true">
        <TableRoot>
            ...
            <TableSection ...>
                ...
                <ViewCell x:Name="_viewCell"
                    Tapped="OnViewCellTapped">
                    <Grid Margin="15,0">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                        </Grid.RowDefinitions>
                        <Label Text="Tap this cell." />
                        <Label x:Name="_target"
                            Grid.Row="1"
                            Text="The cell has changed size."
                            IsVisible="false" />
                    </Grid>
                </ViewCell>
            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>

```

When the `ViewCell` is tapped, the `OnViewCellTapped` event handler is executed:

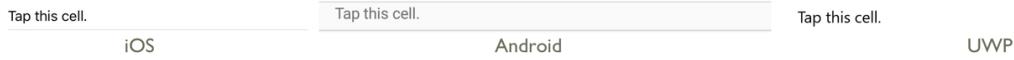
```

void OnViewCellTapped(object sender, EventArgs e)
{
    _target.IsVisible = !_target.IsVisible;
    _viewCell.ForceUpdateSize();
}

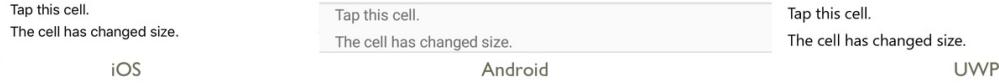
```

The `OnViewCellTapped` event handler shows or hides the second `Label` in the `ViewCell`, and explicitly updates the cell's size by calling the `Cell.ForceUpdateSize` method.

The following screenshots show the cell prior to being tapped upon:



The following screenshots show the cell after being tapped upon:



### IMPORTANT

There is a strong possibility of performance degradation if this feature is overused.

## Related Links

- [TableView \(sample\)](#)

# Text in Xamarin.Forms

11/1/2018 • 2 minutes to read • [Edit Online](#)

Using Xamarin.Forms to enter or display text.

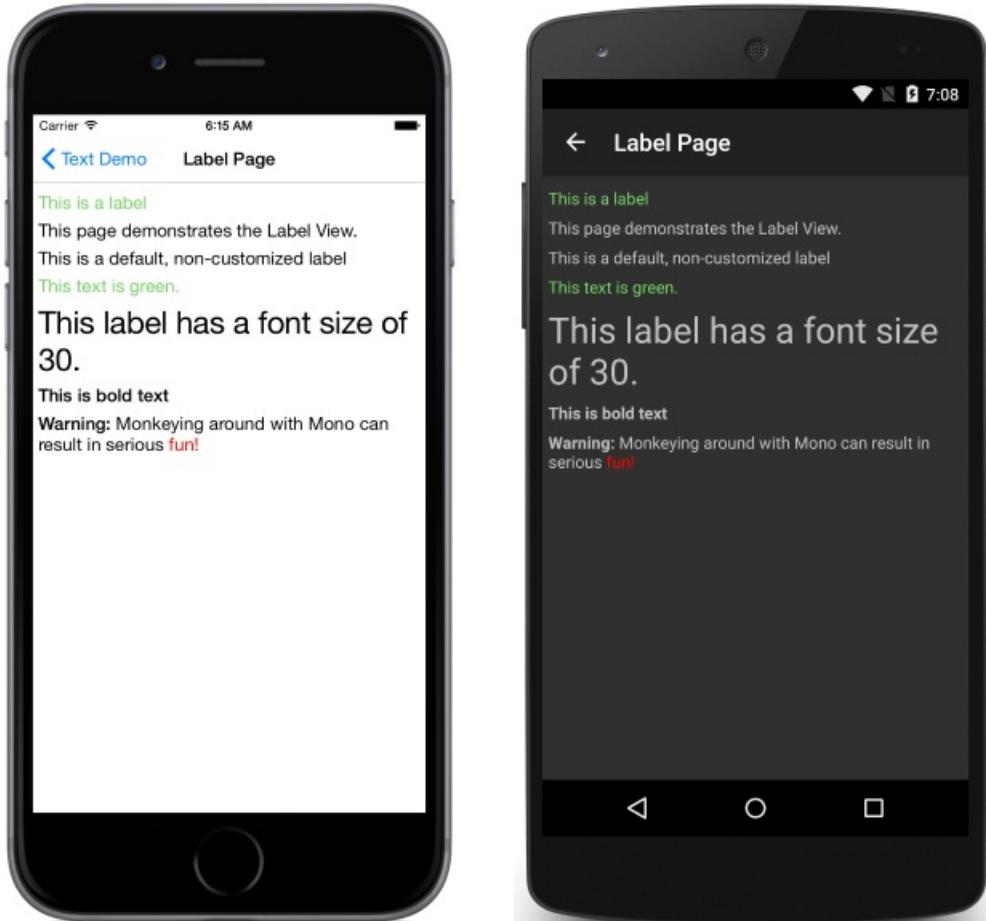
Xamarin.Forms has three primary views for working with text:

- **Label** — for presenting single or multi-line text. Can show text with multiple formatting options in the same line.
- **Entry** — for entering text that is only one line. Entry has a password mode.
- **Editor** — for entering text that could take more than one line.

Text appearance can be changed using built-in or custom [styles](#) and some controls support custom [fonts](#).

## Label

The `Label` view is used to display text. It can show multiple lines of text or a single line of text. `Label` can present text with multiple formatting options used in inline. The label view can wrap or truncate text when it can't fit on one line.

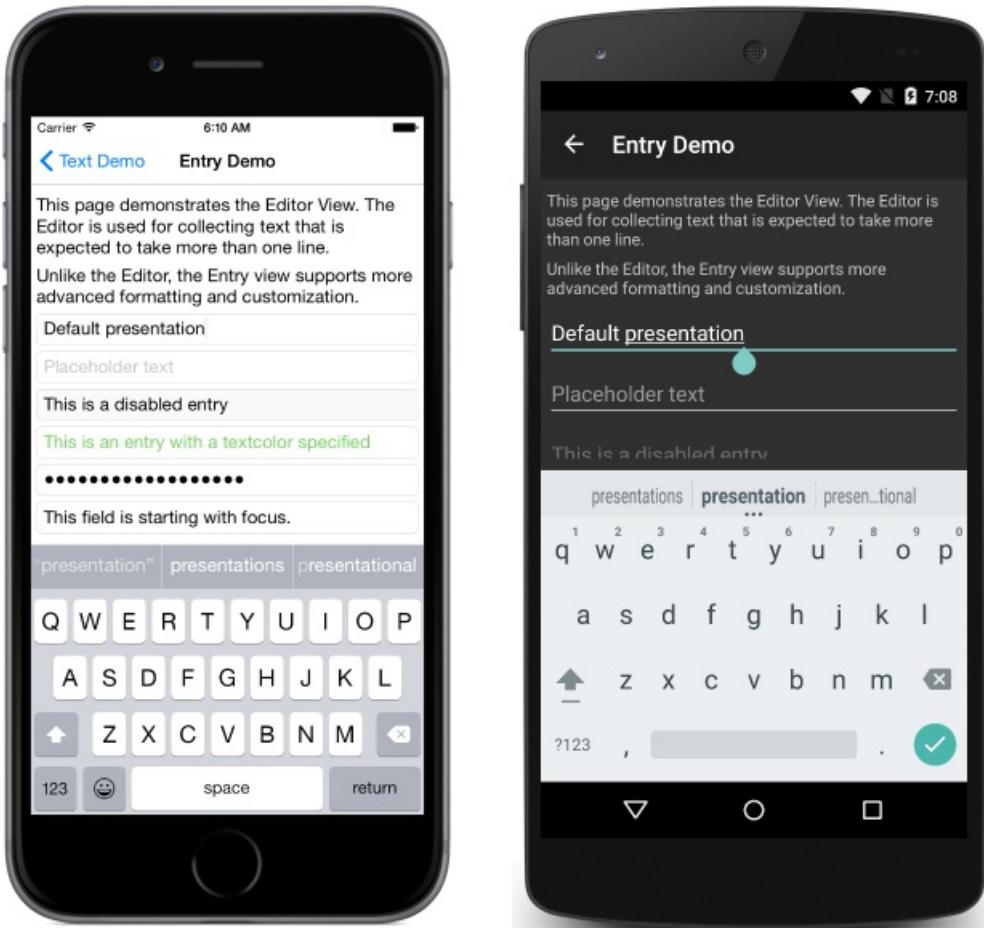


See the [Label](#) article for more detailed information.

For information on customizing the font used in a label, see [Fonts](#).

## Entry

`Entry` is used to accept single-line text input. `Entry` offers control over colors and fonts. `Entry` has a password mode and can show placeholder text until text is entered.

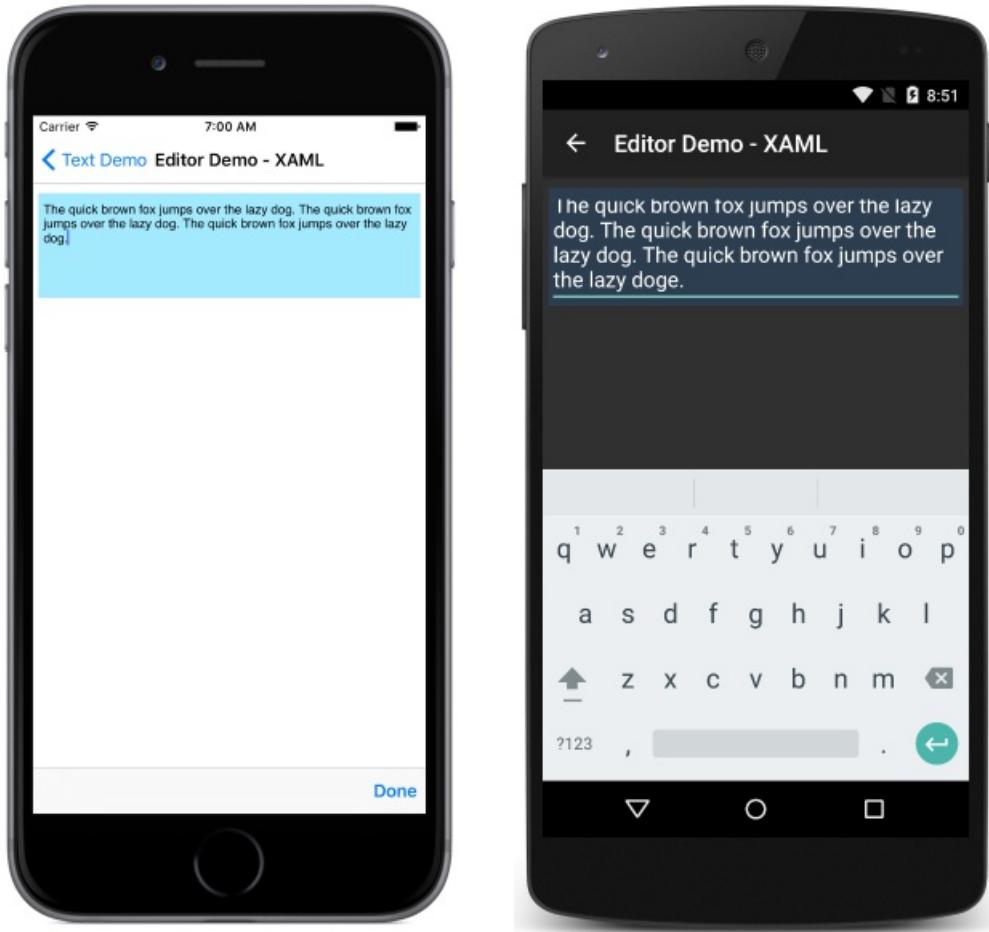


See the [Entry](#) article for more information.

Note that, unlike `Label`, `Entry` cannot have custom font settings.

## Editor

`Editor` is used to accept multi-line text input. `Editor` offers control over colors and fonts.



See the [Editor](#) article for more information.

## Fonts

The `Label` control supports different font settings using the built-in fonts on each platform or custom fonts included with your app. See the [Fonts](#) article for more detailed information.

## Styles

Refer to [working with styles](#) to learn how to set up font, `color`, and other display properties that apply across multiple controls.

## Related Links

- [Text \(sample\)](#)

# Xamarin.Forms Label

10/30/2018 • 8 minutes to read • [Edit Online](#)

## Display text in Xamarin.Forms

The `Label` view is used for displaying text, both single and multi-line. Labels can have text decorations, colored text, and use custom fonts (families, sizes, and options).

## Text decorations

Underline and strikethrough text decorations can be applied to `Label` instances by setting the `Label.TextDecorations` property to one or more `TextDecorations` enumeration members:

- `None`
- `Underline`
- `Strikethrough`

The following XAML example demonstrates setting the `Label.TextDecorations` property:

```
<Label Text="This is underlined text." TextDecorations="Underline" />
<Label Text="This is text with strikethrough." TextDecorations="Strikethrough" />
<Label Text="This is underlined text with strikethrough." TextDecorations="Underline, Strikethrough" />
```

The equivalent C# code is:

```
var underlineLabel = new Label { Text = "This is underlined text.", TextDecorations =
TextDecorations.Underline };
var strikethroughLabel = new Label { Text = "This is text with strikethrough.", TextDecorations =
TextDecorations.Strikethrough };
var bothLabel = new Label { Text = "This is underlined text with strikethrough.", TextDecorations =
TextDecorations.Underline | TextDecorations.Strikethrough };
```

The following screenshots show the `TextDecorations` enumeration members applied to `Label` instances:

This is underlined text.

~~This is text with strikethrough.~~

This is underlined text with strikethrough. ~~This is underlined text with strikethrough.~~

iOS

This is underlined text.

~~This is text with strikethrough.~~

This is underlined text with strikethrough. ~~This is underlined text with strikethrough.~~

Android

This is underlined text.

~~This is text with strikethrough.~~

This is underlined text with strikethrough. ~~This is underlined text with strikethrough.~~

UWP

### NOTE

Text decorations can also be applied to `Span` instances. For more information about the `Span` class, see [Formatted Text](#).

## Colors

Labels can be set to use a custom text color via the bindable `TextColor` property.

Special care is necessary to ensure that colors will be usable on each platform. Because each platform has different defaults for text and background colors, you'll need to be careful to pick a default that works on each.

The following XAML example sets the text color of a `Label`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="TextSample.LabelPage"
    Title="Label Demo">
    <StackLayout Padding="5,10">
        <Label TextColor="#77d065" FontSize = "20" Text="This is a green label." />
    </StackLayout>
</ContentPage>

```

The equivalent C# code is:

```

public partial class LabelPage : ContentPage
{
    public LabelPage ()
    {
        InitializeComponent ();

        var layout = new StackLayout { Padding = new Thickness(5,10) };
        var label = new Label { Text="This is a green label.", TextColor = Color.FromHex("#77d065"), FontSize
= 20 };
        layout.Children.Add(label);
        this.Content = layout;
    }
}

```

The following screenshots show the result of setting the `TextColor` property:



For more information about colors, see [Colors](#).

## Fonts

For more information about specifying fonts on a `Label`, see [Fonts](#).

## Truncation and wrapping

Labels can be set to handle text that can't fit on one line in one of several ways, exposed by the `LineBreakMode` property. `LineBreakMode` is an enumeration with the following values:

- **HeadTruncation** – truncates the head of the text, showing the end.
- **CharacterWrap** – wraps text onto a new line at a character boundary.
- **MiddleTruncation** – displays the beginning and end of the text, with the middle replaced by an ellipsis.
- **NoWrap** – does not wrap text, displaying only as much text as can fit on one line.
- **TailTruncation** – shows the beginning of the text, truncating the end.
- **WordWrap** – wraps text at the word boundary.

## Displaying a specific number of lines

The number of lines displayed by a `Label` can be specified by setting the `Label.MaxLines` property to a `int` value:

- When `MaxLines` is 0, the `Label` respects the value of the `LineBreakMode` property to either show just one line,

possibly truncated, or all lines with all text.

- When `MaxLines` is 1, the result is identical to setting the `LineBreakMode` property to `NoWrap`, `HeadTruncation`, `MiddleTruncation`, or `TailTruncation`. However, the `Label` will respect the value of the `LineBreakMode` property with regard to placement of an ellipsis, if applicable.
- When `MaxLines` is greater than 1, the `Label` will display up to the specified number of lines, while respecting the value of the `LineBreakMode` property with regard to placement of an ellipsis, if applicable. However, setting the `MaxLines` property to a value greater than 1 has no effect if the `LineBreakMode` property is set to `NoWrap`.

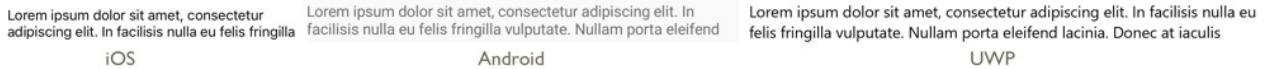
The following XAML example demonstrates setting the `MaxLines` property on a `Label`:

```
<Label Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus."
      LineBreakMode="WordWrap"
      MaxLines="2" />
```

The equivalent C# code is:

```
var label =
{
    Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.", LineBreakMode = LineBreakMode.WordWrap,
    MaxLines = 2
};
```

The following screenshots show the result of setting the `MaxLines` property to 2, when the text is long enough to occupy more than 2 lines:



iOS                    Android                    UWP

## Formatted text

Labels expose a `FormattedText` property which allows the presentation of text with multiple fonts and colors in the same view.

The `FormattedText` property is of type `FormattedString`, which comprises one or more `Span` instances, set via the `Spans` property. The following `Span` properties can be used to set visual appearance:

- `BackgroundColor` – the color of the span background.
- `Font` – the font for the text in the span.
- `FontAttributes` – the font attributes for the text in the span.
- `FontFamily` – the font family to which the font for the text in the span belongs.
- `FontSize` – the size of the font for the text in the span.
- `ForegroundColor` – the color for the text in the span. This property is obsolete and has been replaced by the `TextColor` property.
- `LineHeight` – the multiplier to apply to the default line height of the span. For more information, see [Line Height](#).
- `Style` – the style to apply to the span.
- `Text` – the text of the span.
- `TextColor` – the color for the text in the span.
- `TextDecorations` – the decorations to apply to the text in the span. For more information, see [Text Decorations](#).

In addition, the `GestureRecognizers` property can be used to define a collection of gesture recognizers that will

respond to gestures on the `Span`.

The following XAML example demonstrates a `FormattedText` property that consists of three `Span` instances:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="TextSample.LabelPage"
    Title="Label Demo - XAML">
<StackLayout Padding="5,10">
    ...
    <Label LineBreakMode="WordWrap">
        <Label.FormattedText>
            <FormattedString>
                <Span Text="Red Bold, " TextColor="Red" FontAttributes="Bold" />
                <Span Text="default, " Style="{DynamicResource BodyStyle}">
                    <Span.GestureRecognizers>
                        <TapGestureRecognizer Command="{Binding TapCommand}" />
                    </Span.GestureRecognizers>
                </Span>
                <Span Text="italic small." FontAttributes="Italic" FontSize="Small" />
            </FormattedString>
        </Label.FormattedText>
    </Label>
</StackLayout>
</ContentPage>
```

The equivalent C# code is:

```
public class LabelPageCode : ContentPage
{
    public LabelPageCode ()
    {
        var layout = new StackLayout{ Padding = new Thickness (5, 10) };
        ...

        var formattedString = new FormattedString ();
        formattedString.Spans.Add (new Span{ Text = "Red bold, ", ForegroundColor = Color.Red, FontAttributes = FontAttributes.Bold });

        var span = new Span { Text = "default, " };
        span.GestureRecognizers.Add(new TapGestureRecognizer { Command = new Command(async () => await DisplayAlert("Tapped", "This is a tapped Span.", "OK")) });
        formattedString.Spans.Add(span);
        formattedString.Spans.Add (new Span { Text = "italic small.", FontAttributes = FontAttributes.Italic, FontSize = Device.GetNamedSize(NamedSize.Small, typeof(Label)) });

        layout.Children.Add (new Label { FormattedText = formattedString });
        this.Content = layout;
    }
}
```

### IMPORTANT

The `Text` property of a `Span` can be set through data binding. For more information, see [Data Binding](#).

Note that a `Span` can also respond to any gestures that are added to the span's `GestureRecognizers` collection. For example, a `TapGestureRecognizer` has been added to the second `Span` in the above code examples. Therefore, when this `Span` is tapped the `TapGestureRecognizer` will respond by executing the `ICommand` defined by the `Command` property. For more information about gesture recognizers, see [Xamarin.Forms Gestures](#).

The following screenshots show the result of setting the `FormattedString` property to three `Span` instances:



## Line height

The vertical height of a `Label` and a `Span` can be customized by setting the `Label.LineHeight` property or `Span.LineHeight` to a `double` value. On iOS and Android these values are multipliers of the original line height, and on the Universal Windows Platform (UWP) the `Label.LineHeight` property value is a multiplier of the label font size.

### NOTE

- On iOS, the `Label.LineHeight` and `Span.LineHeight` properties change the line height of text that fits on a single line, and text that wraps onto multiple lines.
- On Android, the `Label.LineHeight` and `Span.LineHeight` properties only change the line height of text that wraps onto multiple lines.
- On UWP, the `Label.LineHeight` property changes the line height of text that wraps onto multiple lines, and the `Span.LineHeight` property has no effect.

The following XAML example demonstrates setting the `LineHeight` property on a `Label`:

```
<Label Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus."
      LineBreakMode="WordWrap"
      LineHeight="1.8" />
```

The equivalent C# code is:

```
var label =
{
    Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.", LineBreakMode = LineBreakMode.WordWrap,
    LineHeight = 1.8
};
```

The following screenshots show the result of setting the `Label.LineHeight` property to 1.8:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.	Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.	Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.
---	---	---

iOS

Android

UWP

The following XAML example demonstrates setting the `LineHeight` property on a `Span`:

```

<Label LineBreakMode="WordWrap">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In a tincidunt sem. Phasellus mollis sit amet turpis in rutrum. Sed aliquam ac urna id scelerisque. ">
                LineHeight="1.8"/>
            <Span Text="Nullam feugiat sodales elit, et maximus nibh vulputate id.">
                LineHeight="1.8" />
        </FormattedString>
    </Label.FormattedText>
</Label>

```

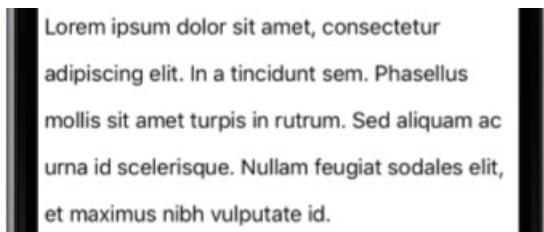
The equivalent C# code is:

```

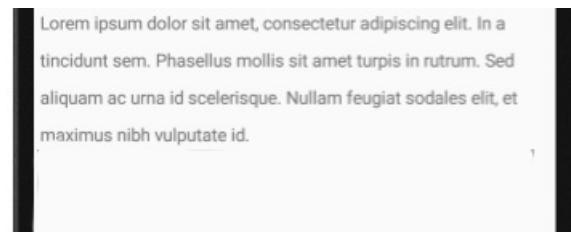
var formattedString = new FormattedString();
formattedString.Spans.Add(new Span
{
    Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In a tincidunt sem. Phasellus mollis sit
    amet turpis in rutrum. Sed aliquam ac urna id scelerisque. ",
    LineHeight = 1.8
});
formattedString.Spans.Add(new Span
{
    Text = "Nullam feugiat sodales elit, et maximus nibh vulputate id.",
    LineHeight = 1.8
});
var label = new Label
{
    FormattedText = formattedString,
    LineBreakMode = LineBreakMode.WordWrap
};

```

The following screenshots show the result of setting the `Span.LineHeight` property to 1.8:



iOS



Android

## Styling labels

The previous sections covered setting `Label` and `Span` properties on a per-instance basis. However, sets of properties can be grouped into one style that is consistently applied to one or many views. This can increase readability of code and make design changes easier to implement. For more information, see [Styles](#).

## Related links

- [Text \(sample\)](#)
- [Creating Mobile Apps with Xamarin.Forms, Chapter 3](#)
- [Label API](#)
- [Span API](#)

# Xamarin.Forms Entry

9/20/2018 • 8 minutes to read • [Edit Online](#)

*Single-line text or password input*

The Xamarin.Forms `Entry` is used for single-line text input. The `Entry`, like the `Editor` view, supports multiple keyboard types. Additionally, the `Entry` can be used as a password field.

## Display Customization

### Setting and Reading Text

The `Entry`, like other text-presenting views, exposes the `Text` property. This property can be used to set and read the text presented by the `Entry`. The following example demonstrates setting the `Text` property in XAML:

```
<Entry Text="I am an Entry" />
```

In C#:

```
var MyEntry = new Entry { Text = "I am an Entry" };
```

To read text, access the `Text` property in C#:

```
var text = MyEntry.Text;
```

#### NOTE

The width of an `Entry` can be defined by setting its `WidthRequest` property. Do not depend on the width of an `Entry` being defined based on the value of its `Text` property.

### Limiting Input Length

The `MaxLength` property can be used to limit the input length that's permitted for the `Entry`. This property should be set to a positive integer:

```
<Entry ... MaxLength="10" />
```

```
var entry = new Entry { ... MaxLength = 10 };
```

A `MaxLength` property value of 0 indicates that no input will be allowed, and a value of `int.MaxValue`, which is the default value for an `Entry`, indicates that there is no effective limit on the number of characters that may be entered.

### Setting the Cursor Position and Text Selection Length

The `CursorPosition` property can be used to return or set the position at which the next character will be inserted into the string stored in the `Text` property:

```
<Entry Text="Cursor position set" CursorPosition="5" />
```

```
var entry = new Entry { Text = "Cursor position set", CursorPosition = 5 };
```

The default value of the `CursorPosition` property is 0, which indicates that text will be inserted at the start of the `Entry`.

In addition, the `SelectionLength` property can be used to return or set the length of text selection within the `Entry`:

```
<Entry Text="Cursor position and selection length set" CursorPosition="2" SelectionLength="10" />
```

```
var entry = new Entry { Text = "Cursor position and selection length set", CursorPosition = 2, SelectionLength = 10 };
```

The default value of the `SelectionLength` property is 0, which indicates that no text is selected.

## Customizing the Keyboard

The keyboard that's presented when users interact with an `Entry` can be set programmatically via the `Keyboard` property, to one of the following properties from the `Keyboard` class:

- `Chat` – used for texting and places where emoji are useful.
- `Default` – the default keyboard.
- `Email` – used when entering email addresses.
- `Numeric` – used when entering numbers.
- `Plain` – used when entering text, without any `KeyboardFlags` specified.
- `Telephone` – used when entering telephone numbers.
- `Text` – used when entering text.
- `Url` – used for entering file paths & web addresses.

This can be accomplished in XAML as follows:

```
<Entry Keyboard="Chat" />
```

The equivalent C# code is:

```
var entry = new Entry { Keyboard = Keyboard.Chat };
```

Examples of each keyboard can be found in our [Recipes](#) repository.

The `Keyboard` class also has a `Create` factory method that can be used to customize a keyboard by specifying capitalization, spellcheck, and suggestion behavior. `KeyboardFlags` enumeration values are specified as arguments to the method, with a customized `Keyboard` being returned. The `KeyboardFlags` enumeration contains the following values:

- `None` – no features are added to the keyboard.
- `CapitalizeSentence` – indicates that the first letter of the first word of each entered sentence will be automatically capitalized.
- `Spellcheck` – indicates that spellcheck will be performed on entered text.

- `Suggestions` – indicates that word completions will be offered on entered text.
- `CapitalizeWord` – indicates that the first letter of each word will be automatically capitalized.
- `CapitalizeCharacter` – indicates that every character will be automatically capitalized.
- `CapitalizeNone` – indicates that no automatic capitalization will occur.
- `All` – indicates that spellcheck, word completions, and sentence capitalization will occur on entered text.

The following XAML code example shows how to customize the default `Keyboard` to offer word completions and capitalize every entered character:

```
<Entry Placeholder="Enter text here">
    <Entry.Keyboard>
        <Keyboard x:FactoryMethod="Create">
            <x:Arguments>
                <KeyboardFlags>Suggestions,CapitalizeCharacter</KeyboardFlags>
            </x:Arguments>
        </Keyboard>
    </Entry.Keyboard>
</Entry>
```

The equivalent C# code is:

```
var entry = new Entry { Placeholder = "Enter text here" };
entry.Keyboard = Keyboard.Create(KeyboardFlags.Suggestions | KeyboardFlags.CapitalizeCharacter);
```

#### Customizing the Return Key

The appearance of the return key on the soft keyboard, which is displayed when an `Entry` has focus, can be customized by setting the `ReturnType` property to a value of the `ReturnType` enumeration:

- `Default` – indicates that no specific return key is required and that the platform default will be used.
- `Done` – indicates a "Done" return key.
- `Go` – indicates a "Go" return key.
- `Next` – indicates a "Next" return key.
- `Search` – indicates a "Search" return key.
- `Send` – indicates a "Send" return key.

The following XAML example shows how to set the return key:

```
<Entry ReturnType="Send" />
```

The equivalent C# code is:

```
var entry = new Entry { ReturnType = ReturnType.Send };
```

#### NOTE

The exact appearance of the return key is dependent upon the platform. On iOS, the return key is a text-based button. However, on the Android and Universal Windows Platforms, the return key is a icon-based button.

When the return key is pressed, the `Completed` event fires and any `ICommand` specified by the `ReturnCommand` property is executed. In addition, any `object` specified by the `ReturnCommandParameter` property will be passed to the `ICommand` as a parameter. For more information about commands, see [The Command Interface](#).

## Enabling and Disabling Spell Checking

The `IsSpellCheckEnabled` property controls whether spell checking is enabled. By default, the property is set to `true`. As the user enters text, misspellings are indicated.

However, for some text entry scenarios, such as entering a username, spell checking provides a negative experience and should be disabled by setting the `IsSpellCheckEnabled` property to `false`:

```
<Entry ... IsSpellCheckEnabled="false" />
```

```
var entry = new Entry { ... IsSpellCheckEnabled = false };
```

### NOTE

When the `IsSpellCheckEnabled` property is set to `false`, and a custom keyboard isn't being used, the native spell checker will be disabled. However, if a `Keyboard` has been set that disables spell checking, such as `Keyboard.Chat`, the `IsSpellCheckEnabled` property is ignored. Therefore, the property cannot be used to enable spell checking for a `Keyboard` that explicitly disables it.

## Enabling and Disabling Text Prediction

The `IsTextPredictionEnabled` property controls whether text prediction and automatic text correction is enabled. By default, the property is set to `true`. As the user enters text, word predictions are presented.

However, for some text entry scenarios, such as entering a username, text prediction and automatic text correction provides a negative experience and should be disabled by setting the `IsTextPredictionEnabled` property to `false`:

```
<Entry ... IsTextPredictionEnabled="false" />
```

```
var entry = new Entry { ... IsTextPredictionEnabled = false };
```

### NOTE

When the `IsTextPredictionEnabled` property is set to `false`, and a custom keyboard isn't being used, text prediction and automatic text correction is disabled. However, if a `Keyboard` has been set that disables text prediction, the `IsTextPredictionEnabled` property is ignored. Therefore, the property cannot be used to enable text prediction for a `Keyboard` that explicitly disables it.

## Setting Placeholder Text

The `Entry` can be set to show placeholder text when it is not storing user input. This is accomplished by setting the `Placeholder` property to a `string`, and is often used to indicate the type of content that is appropriate for the `Entry`. In addition, the placeholder text color can be controlled by setting the `PlaceholderColor` property to a `Color`:

```
<Entry Placeholder="Username" PlaceholderColor="Olive" />
```

```
var entry = new Entry { Placeholder = "Username", PlaceholderColor = Color.Olive };
```

## Password Fields

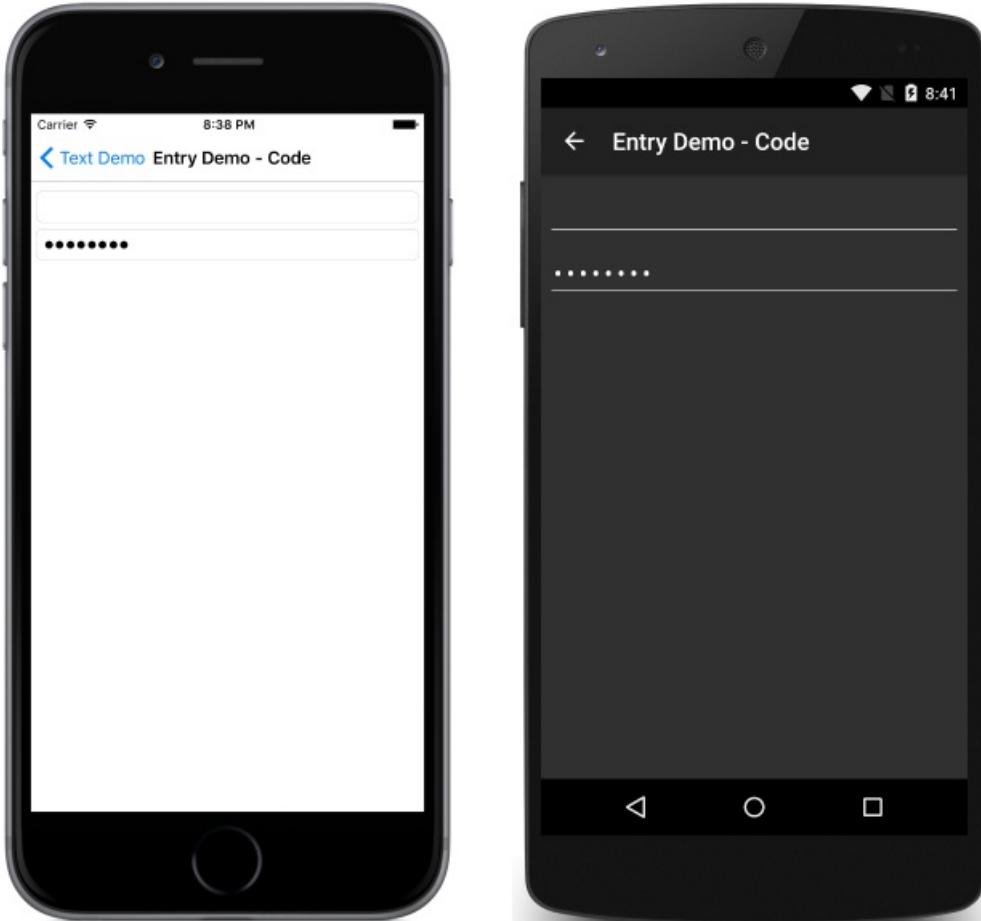
`Entry` provides the `IsPassword` property. When `IsPassword` is `true`, the contents of the field will be presented as black circles:

In XAML:

```
<Entry IsPassword="true" />
```

In C#:

```
var MyEntry = new Entry { IsPassword = true };
```



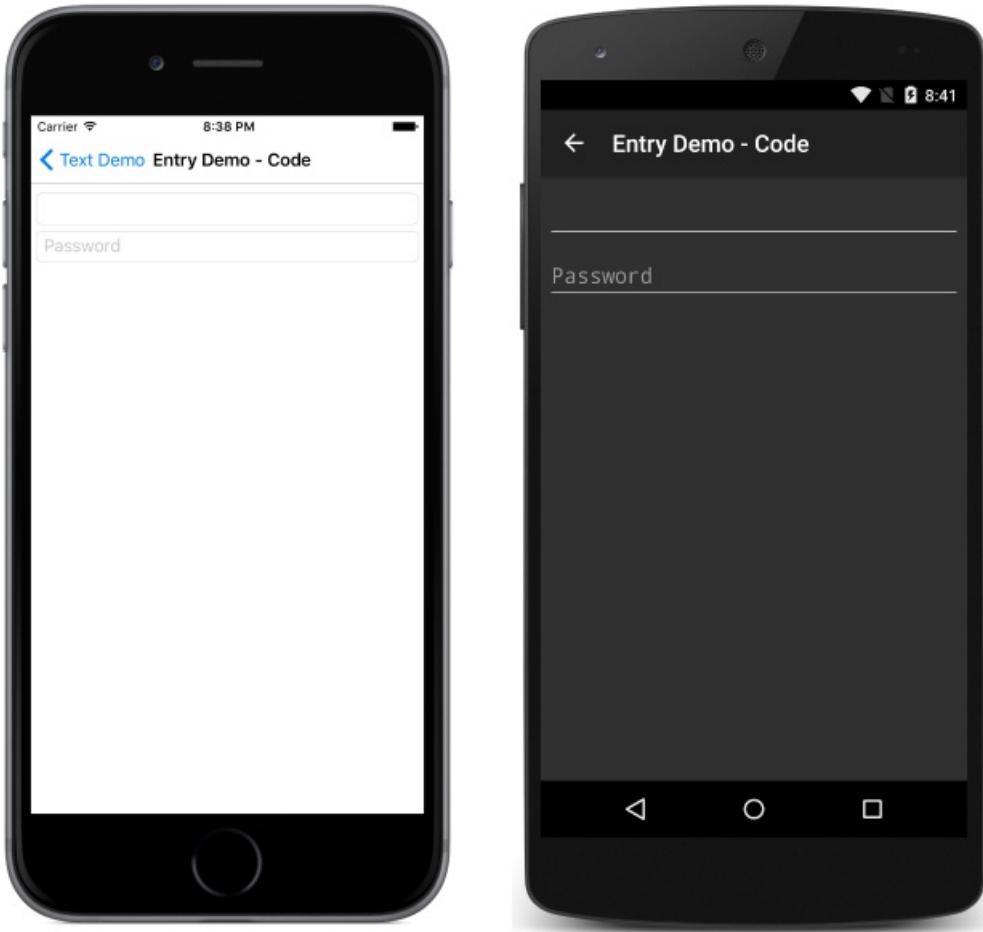
Placeholders may be used with instances of `Entry` that are configured as password fields:

In XAML:

```
<Entry IsPassword="true" Placeholder="Password" />
```

In C#:

```
var MyEntry = new Entry { IsPassword = true, Placeholder = "Password" };
```



## Colors

Entry can be set to use a custom background and text colors via the following bindable properties:

- **TextColor** – sets the color of the text.
- **BackgroundColor** – sets the color shown behind the text.

Special care is necessary to ensure that colors will be usable on each platform. Because each platform has different defaults for text and background colors, you'll often need to set both if you set one.

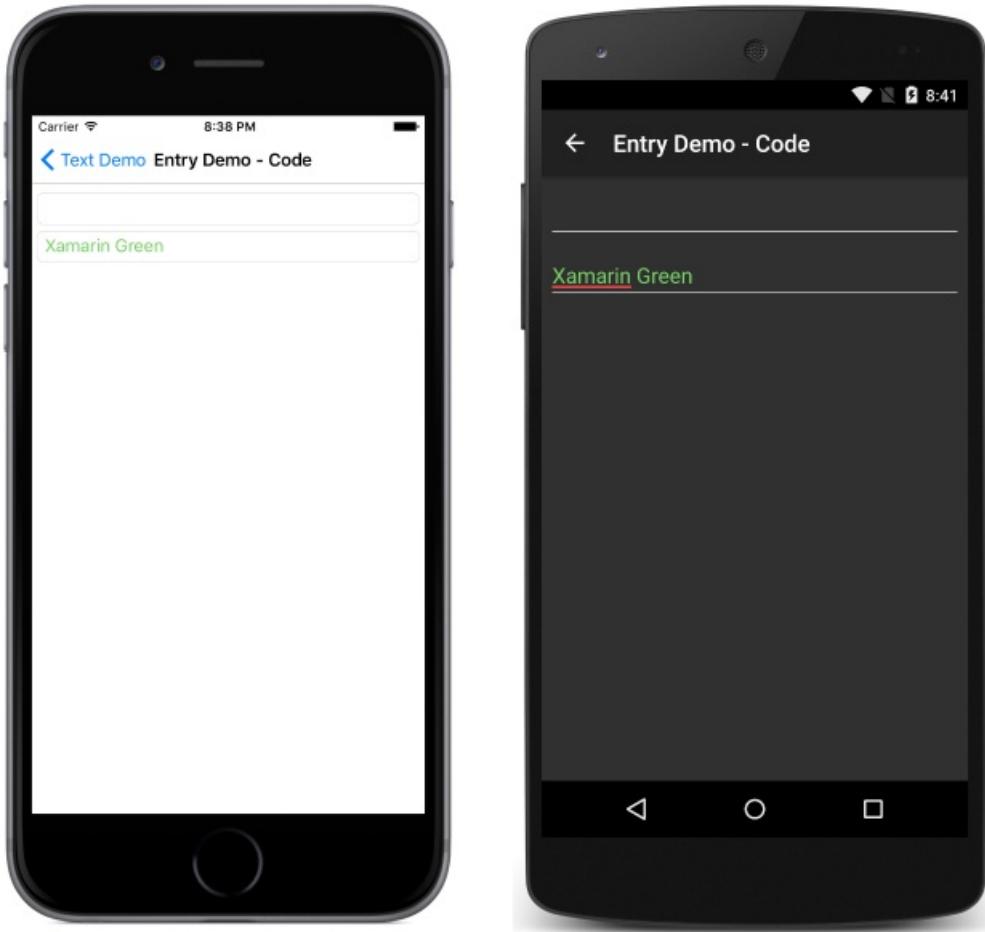
Use the following code to set the text color of an entry:

In XAML:

```
<Entry TextColor="Green" />
```

In C#:

```
var entry = new Entry();
entry.TextColor = Color.Green;
```



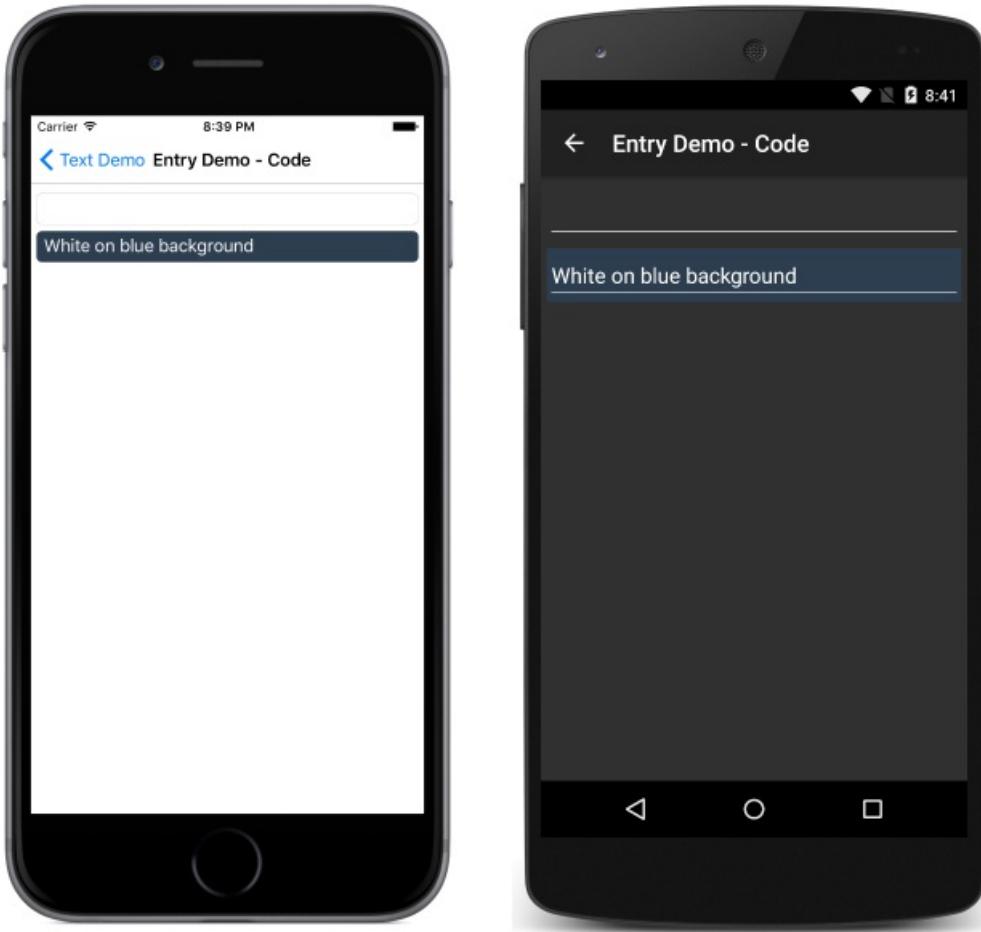
Note that the placeholder is not affected by the specified `TextColor`.

To set the background color in XAML:

```
<Entry BackgroundColor="#2c3e50" />
```

In C#:

```
var entry = new Entry();
entry.BackgroundColor = Color.FromHex("#2c3e50");
```



Be careful to make sure that the background and text colors you choose are usable on each platform and don't obscure any placeholder text.

## Events and Interactivity

Entry exposes two events:

- `TextChanged` – raised when the text changes in the entry. Provides the text before and after the change.
- `Completed` – raised when the user has ended input by pressing the return key on the keyboard.

### Completed

The `Completed` event is used to react to the completion of an interaction with an Entry. `Completed` is raised when the user ends input with a field by pressing the return key on the keyboard. The handler for the event is a generic event handler, taking the sender and `EventArgs`:

```
void Entry_Completed (object sender, EventArgs e)
{
    var text = ((Entry)sender).Text; //cast sender to access the properties of the Entry
}
```

The completed event can be subscribed to in XAML:

```
<Entry Completed="Entry_Completed" />
```

and C#:

```
var entry = new Entry ();
entry.Completed += Entry_Completed;
```

After the `Completed` event fires, any `ICommand` specified by the `ReturnCommand` property is executed, with the `object` specified by the `ReturnCommandParameter` property being passed to the `ICommand`.

## TextChanged

The `TextChanged` event is used to react to a change in the content of a field.

`TextChanged` is raised whenever the `Text` of the `Entry` changes. The handler for the event takes an instance of `TextChangedEventArgs`. `TextChangedEventArgs` provides access to the old and new values of the `Entry` `Text` via the `OldTextValue` and `NewTextValue` properties:

```
void Entry_TextChanged (object sender, TextChangedEventArgs e)
{
    var oldText = e.OldTextValue;
    var newText = e.NewTextValue;
}
```

The `TextChanged` event can be subscribed to in XAML:

```
<Entry TextChanged="Entry_TextChanged" />
```

and C#:

```
var entry = new Entry ();
entry.TextChanged += Entry_TextChanged;
```

## Related Links

- [Text \(sample\)](#)
- [Entry API](#)

# Xamarin.Forms Editor

9/20/2018 • 6 minutes to read • [Edit Online](#)

## Multi-line text input

The `Editor` control is used to accept multi-line input. This article covers:

- **Customization** – keyboard and color options.
- **Interactivity** – events that can be listened for to provide interactivity.

## Customization

### Setting and Reading Text

The `Editor`, like other text-presenting views, exposes the `Text` property. This property can be used to set and read the text presented by the `Editor`. The following example demonstrates setting the `Text` property in XAML:

```
<Editor Text="I am an Editor" />
```

In C#:

```
var MyEditor = new Editor { Text = "I am an Editor" };
```

To read text, access the `Text` property in C#:

```
var text = MyEditor.Text;
```

### Limiting Input Length

The `MaxLength` property can be used to limit the input length that's permitted for the `Editor`. This property should be set to a positive integer:

```
<Editor ... MaxLength="10" />
```

```
var editor = new Editor { ... MaxLength = 10 };
```

A `MaxLength` property value of 0 indicates that no input will be allowed, and a value of `int.MaxValue`, which is the default value for an `Editor`, indicates that there is no effective limit on the number of characters that may be entered.

### Auto-Sizing an Editor

An `Editor` can be made to auto-size to its content by setting the `Editor.AutoSize` property to `TextChanges`, which is a value of the `EditoAutoSizeOption` enumeration. This enumeration has two values:

- `Disabled` indicates that automatic resizing is disabled, and is the default value.
- `TextChanges` indicates that automatic resizing is enabled.

This can be accomplished in code as follows:

```
<Editor Text="Enter text here" AutoSize="TextChanges" />
```

```
var editor = new Editor { Text = "Enter text here", AutoSize = EditorAutoSizeOption.TextChanges };
```

When auto-resizing is enabled, the height of the `Editor` will increase when the user fills it with text, and the height will decrease as the user deletes text.

#### NOTE

An `Editor` will not auto-size if the `HeightRequest` property has been set.

## Customizing the Keyboard

The keyboard that's presented when users interact with an `Editor` can be set programmatically via the `Keyboard` property, to one of the following properties from the `Keyboard` class:

- `Chat` – used for texting and places where emoji are useful.
- `Default` – the default keyboard.
- `Email` – used when entering email addresses.
- `Numeric` – used when entering numbers.
- `Plain` – used when entering text, without any `KeyboardFlags` specified.
- `Telephone` – used when entering telephone numbers.
- `Text` – used when entering text.
- `Url` – used for entering file paths & web addresses.

This can be accomplished in XAML as follows:

```
<Editor Keyboard="Chat" />
```

The equivalent C# code is:

```
var editor = new Editor { Keyboard = Keyboard.Chat };
```

Examples of each keyboard can be found in our [Recipes](#) repository.

The `Keyboard` class also has a `Create` factory method that can be used to customize a keyboard by specifying capitalization, spellcheck, and suggestion behavior. `KeyboardFlags` enumeration values are specified as arguments to the method, with a customized `Keyboard` being returned. The `KeyboardFlags` enumeration contains the following values:

- `None` – no features are added to the keyboard.
- `CapitalizeSentence` – indicates that the first letter of the first word of each entered sentence will be automatically capitalized.
- `Spellcheck` – indicates that spellcheck will be performed on entered text.
- `Suggestions` – indicates that word completions will be offered on entered text.
- `CapitalizeWord` – indicates that the first letter of each word will be automatically capitalized.
- `CapitalizeCharacter` – indicates that every character will be automatically capitalized.
- `CapitalizeNone` – indicates that no automatic capitalization will occur.
- `All` – indicates that spellcheck, word completions, and sentence capitalization will occur on entered text.

The following XAML code example shows how to customize the default `Keyboard` to offer word completions and capitalize every entered character:

```
<Editor>
  <Editor.Keyboard>
    <Keyboard x:FactoryMethod="Create">
      <x:Arguments>
        <KeyboardFlags>Suggestions,CapitalizeCharacter</KeyboardFlags>
      </x:Arguments>
    </Keyboard>
  </Editor.Keyboard>
</Editor>
```

The equivalent C# code is:

```
var editor = new Editor();
editor.Keyboard = Keyboard.Create(KeyboardFlags.Suggestions | KeyboardFlags.CapitalizeCharacter);
```

## Enabling and Disabling Spell Checking

The `IsSpellCheckEnabled` property controls whether spell checking is enabled. By default, the property is set to `true`. As the user enters text, misspellings are indicated.

However, for some text entry scenarios, such as entering a username, spell checking provides a negative experience and so should be disabled by setting the `IsSpellCheckEnabled` property to `false`:

```
<Editor ... IsSpellCheckEnabled="false" />
```

```
var editor = new Editor { ... IsSpellCheckEnabled = false };
```

### NOTE

When the `IsSpellCheckEnabled` property is set to `false`, and a custom keyboard isn't being used, the native spell checker will be disabled. However, if a `Keyboard` has been set that disables spell checking, such as `Keyboard.Chat`, the `IsSpellCheckEnabled` property is ignored. Therefore, the property cannot be used to enable spell checking for a `Keyboard` that explicitly disables it.

## Setting Placeholder Text

The `Editor` can be set to show placeholder text when it is not storing user input. This is accomplished by setting the `Placeholder` property to a `string`, and is often used to indicate the type of content that is appropriate for the `Editor`. In addition, the placeholder text color can be controlled by setting the `PlaceholderColor` property to a `Color`:

```
<Editor Placeholder="Enter text here" PlaceholderColor="Olive" />
```

```
var editor = new Editor { Placeholder = "Enter text here", PlaceholderColor = Color.Olive };
```

## Colors

`Editor` can be set to use a custom background color via the `BackgroundColor` property. Special care is necessary to ensure that colors will be usable on each platform. Because each platform has different defaults for text color, you may need to set a custom background color for each platform. See [Working with Platform Tweaks](#) for more

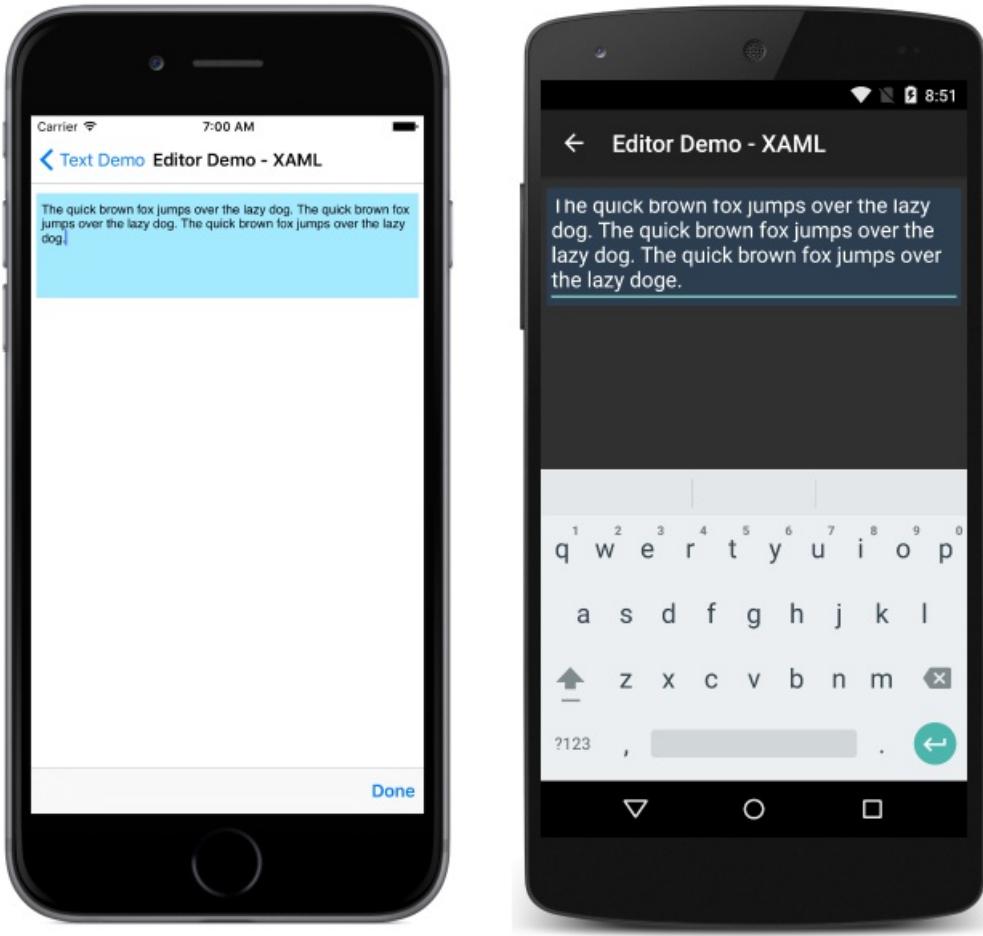
information about optimizing the UI for each platform.

In C#:

```
public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        //dark blue on UWP & Android, light blue on iOS
        var editor = new Editor { BackgroundColor = Device.RuntimePlatform == Device.iOS ?
Color.FromHex("#A4EAFF") : Color.FromHex("#2c3e50") };
        layout.Children.Add(editor);
    }
}
```

In XAML:

```
<?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="TextSample.EditorPage"
    Title="Editor Demo">
    <ContentPage.Content>
        <StackLayout Padding="5,10">
            <Editor>
                <Editor.BackgroundColor>
                    <OnPlatform x:TypeArguments="x:Color">
                        <On Platform="iOS" Value="#a4eaff" />
                        <On Platform="Android, UWP" Value="#2c3e50" />
                    </OnPlatform>
                </Editor.BackgroundColor>
            </Editor>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```



Make sure that the background and text colors you choose are usable on each platform and don't obscure any placeholder text.

## Interactivity

`Editor` exposes two events:

- `TextChanged` – raised when the text changes in the editor. Provides the text before and after the change.
- `Completed` – raised when the user has ended input by pressing the return key on the keyboard.

### Completed

The `Completed` event is used to react to the completion of an interaction with an `Editor`. `Completed` is raised when the user ends input with a field by entering the return key on the keyboard. The handler for the event is a generic event handler, taking the sender and `EventArgs`:

```
void EditorCompleted (object sender, EventArgs e)
{
    var text = ((Editor)sender).Text; // sender is cast to an Editor to enable reading the `Text` property of
                                    // the view.
}
```

The completed event can be subscribed to in code and XAML:

In C#:

```

public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        var editor = new Editor ();
        editor.Completed += EditorCompleted;
        layout.Children.Add(editor);
    }
}

```

In XAML:

```

<?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="TextSample.EditorPage"
    Title="Editor Demo">
    <ContentPage.Content>
        <StackLayout Padding="5,10">
            <Editor Completed="EditorCompleted" />
        </StackLayout>
    </ContentPage.Content>
</Contentpage>

```

## TextChanged

The `TextChanged` event is used to react to a change in the content of a field.

`TextChanged` is raised whenever the `Text` of the `Editor` changes. The handler for the event takes an instance of `TextChangedEventArgs`. `TextChangedEventArgs` provides access to the old and new values of the `Editor` `Text` via the `OldTextValue` and `NewTextValue` properties:

```

void EditorTextChanged (object sender, TextChangedEventArgs e)
{
    var oldText = e.OldTextValue;
    var newText = e.NewTextValue;
}

```

The completed event can be subscribed to in code and XAML:

In code:

```

public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        var editor = new Editor ();
        editor.TextChanged += EditorTextChanged;
        layout.Children.Add(editor);
    }
}

```

In XAML:

```
<?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="TextSample.EditorPage"
  Title="Editor Demo">
  <ContentPage.Content>
    <StackLayout Padding="5,10">
      <Editor TextChanged="EditorTextChanged" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

## Related Links

- [Text \(sample\)](#)
- [Editor API](#)

# Fonts in Xamarin.Forms

7/23/2018 • 5 minutes to read • [Edit Online](#)

This article describes how Xamarin.Forms lets you specify font attributes (including weight and size) on controls that display text. Font information can be [specified in code](#) or [specified in XAML](#). It is also possible to use a [custom font](#).

## Setting Font in Code

Use the three font-related properties of any controls that display text:

- **FontFamily** – the `string` font name.
- **FontSize** – the font size as a `double`.
- **FontAttributes** – a string specifying style information like *Italic* and **Bold** (using the `FontAttributes` enumeration in C#).

This code shows how to create a label and specify the font size and weight to display:

```
var about = new Label {
    FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label)),
    FontAttributes = FontAttributes.Bold,
    Text = "Medium Bold Font"
};
```

### Font Size

The `FontSize` property can be set to a double value, for instance:

```
label.FontSize = 24;
```

You can also use the `NamedSize` enumeration which has four built-in options; Xamarin.Forms chooses the best size for each platform.

- **Micro**
- **Small**
- **Medium**
- **Large**

The `NamedSize` enumeration can be used wherever a `FontSize` can be specified using the `Device.GetNamedSize` method to convert the value to a `double`:

```
label.FontSize = Device.GetNamedSize(NamedSize.Small, typeof(Label));
```

### Font Attributes

Font styles such as **bold** and *italic* can be set on the `FontAttributes` property. The following values are currently supported:

- **None**
- **Bold**
- **Italic**

The `FontAttribute` enumeration can be used as follows (you can specify a single attribute or `OR` them together):

```
label.FontAttributes = FontAttributes.Bold | FontAttributes.Italic;
```

## Setting Font Info Per Platform

Alternatively, the `Device.RuntimePlatform` property can be used to set different font names on each platform, as demonstrated in this code:

```
label.FontFamily = Device.RuntimePlatform == Device.iOS ? "Lobster-Regular" :  
    Device.RuntimePlatform == Device.Android ? "Lobster-Regular.ttf#Lobster-Regular" : "Assets/Fonts/Lobster-  
    Regular.ttf#Lobster",  
label.FontSize = Device.RuntimePlatform == Device.iOS ? 24 :  
    Device.RuntimePlatform == Device.Android ? Device.GetNamedSize(NamedSize.Medium, label) :  
    Device.GetNamedSize(NamedSize.Large, label);
```

A good source of font information for iOS is [iosfonts.com](http://iosfonts.com).

## Setting the Font in XAML

Xamarin.Forms controls that display text all have a `Font` property that can be set in XAML. The simplest way to set the font in XAML is to use the named size enumeration values, as shown in this example:

```
<Label Text="Login" FontSize="Large"/>  
<Label Text="Instructions" FontSize="Small"/>
```

There is a built-in converter for the `Font` property that allows all font settings to be expressed as a string value in XAML. The following examples show how you can specify font attributes and sizes in XAML:

```
<Label Text="Italics are supported" FontAttributes="Italic" />  
<Label Text="Biggest NamedSize" FontSize="Large" />  
<Label Text="Use size 72" FontSize="72" />
```

To specify multiple `Font` settings, combine the required settings into a single `Font` attribute string. The font attribute string should be formatted as `"[font-face],[attributes],[size]"`. The order of the parameters is important, all parameters are optional, and multiple `attributes` can be specified, for example:

```
<Label Text="Small bold text" Font="Bold, Micro" />  
<Label Text="Medium custom font" Font="MarkerFelt-Thin, 42" />  
<Label Text="Really big bold and italic text" Font="Bold, Italic, 72" />
```

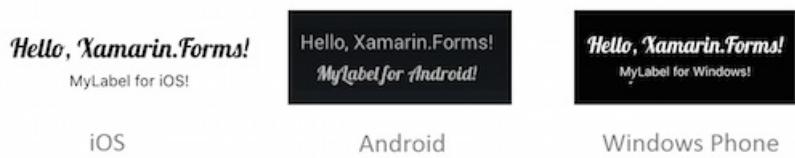
`Device.RuntimePlatform` can also be used in XAML to render a different font on each platform. The example below uses a custom font face on iOS (`MarkerFelt-Thin`) and specifies only size/attributes on the other platforms:

```
<Label Text="Hello Forms with XAML">  
    <Label.FontFamily>  
        <OnPlatform x:TypeArguments="x:String">  
            <On Platform="iOS" Value="MarkerFelt-Thin" />  
            <On Platform="Android" Value="Lobster-Regular.ttf#Lobster-Regular" />  
            <On Platform="UWP" Value="Assets/Fonts/Lobster-Regular.ttf#Lobster" />  
        </OnPlatform>  
    </Label.FontFamily>  
</Label>
```

When specifying a custom font face, it is always a good idea to use `OnPlatform`, as it is difficult to find a font that is available on all platforms.

## Using a Custom Font

Using a font other than the built-in typefaces requires some platform-specific coding. This screenshot shows the custom font **Lobster** from [Google's open-source fonts](#) rendered using Xamarin.Forms.



The steps required for each platform are outlined below. When including custom font files with an application, be sure to verify that the font's license allows for distribution.

### iOS

It is possible to display a custom font by first ensuring that it is loaded, then referring to it by name using the Xamarin.Forms `Font` methods. Follow the instructions in [this blog post](#):

1. Add the font file with **Build Action: BundleResource**, and
2. Update the **Info.plist** file (**FONTS PROVIDED BY APPLICATION**, or `UIAppFonts`, key), then
3. Refer to it by name wherever you define a font in Xamarin.Forms!

```
new Label
{
    Text = "Hello, Forms!",
    FontFamily = Device.RuntimePlatform == Device.iOS ? "Lobster-Regular" : null // set only for iOS
}
```

### Android

Xamarin.Forms for Android can reference a custom font that has been added to the project by following a specific naming standard. First add the font file to the **Assets** folder in the application project and set *Build Action: AndroidAsset*. Then use the full path and *Font Name* separated by a hash (#) as the font name in Xamarin.Forms, as the code snippet below demonstrates:

```
new Label
{
    Text = "Hello, Forms!",
    FontFamily = Device.RuntimePlatform == Device.Android ? "Lobster-Regular.ttf#Lobster-Regular" : null // set
only for Android
}
```

### Windows

Xamarin.Forms for Windows platforms can reference a custom font that has been added to the project by following a specific naming standard. First add the font file to the **/Assets/Fonts/** folder in the application project and set the *Build Action: Content*. Then use the full path and font filename, followed by a hash (#) and the *Font Name*, as the code snippet below demonstrates:

```
new Label
{
    Text = "Hello, Forms!",
    FontFamily = Device.RuntimePlatform == Device.UWP ? "Assets/Fonts/Lobster-Regular.ttf#Lobster" : null // set only for UWP apps
}
```

#### NOTE

Note that the font file name and font name may be different. To discover the font name on Windows, right-click the .ttf file and select **Preview**. The font name can then be determined from the preview window.

The common code for the application is now complete. Platform-specific phone dialer code will now be implemented as a [DependencyService](#).

#### XAML

You can also use `Device.RuntimePlatform` in XAML to render a custom font:

```
<Label Text="Hello Forms with XAML">
    <Label.FontFamily>
        <OnPlatform x:TypeArguments="x:String">
            <On Platform="iOS" Value="Lobster-Regular" />
            <On Platform="Android" Value="Lobster-Regular.ttf#Lobster-Regular" />
            <On Platform="UWP" Value="Assets/Fonts/Lobster-Regular.ttf#Lobster" />
        </OnPlatform>
    </Label.FontFamily>
</Label>
```

## Summary

Xamarin.Forms provides simple default settings to let you size text easily for all supported platforms. It also lets you specify font face and size – even differently for each platform – when more fine-grained control is required.

Font information can also be specified in XAML using correctly formatted font attributes.

## Related Links

- [FontsSample](#)
- [Text \(sample\)](#)

# Xamarin.Forms Text Styles

7/12/2018 • 2 minutes to read • [Edit Online](#)

## *Styling text in Xamarin.Forms*

Styles can be used to adjust the appearance of labels, entries, and editors. Styles can be defined once and used by many views, but a style can only be used with views of one type. Styles can be given a `Key` and applied selectively using a specific control's `style` property.

## Built-In Styles

Xamarin.Forms includes several [built-in](#) styles for common scenarios:

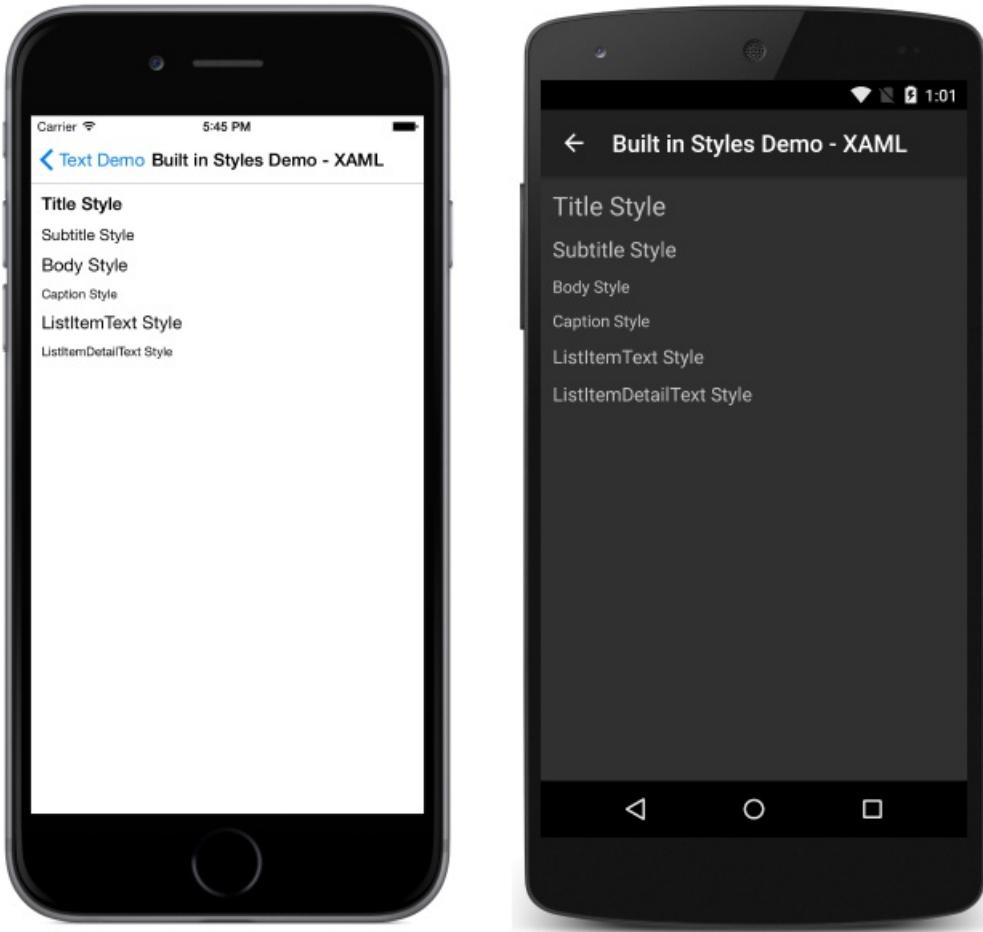
- `BodyStyle`
- `CaptionStyle`
- `ListItemDetailTextStyle`
- `ListItemTextStyle`
- `SubtitleStyle`
- `TitleStyle`

To apply one of the built-in styles, use the `DynamicResource` markup extension to specify the style:

```
<Label Text="I'm a Title" Style="{DynamicResource TitleStyle}">
```

In C#, built-in styles are selected from `Device.Styles`:

```
label.Style = Device.Styles.TitleStyle;
```



## Custom Styles

Styles consist of setters and setters consist of properties and the values the properties will be set to.

In C#, a custom style for a label with red text of size 30 would be defined as follows:

```
var LabelStyle = new Style (typeof(Label)) {
    Setters = {
        new Setter {Property = Label.TextColorProperty, Value = Color.Red},
        new Setter {Property = Label.FontSizeProperty, Value = 30}
    }
};

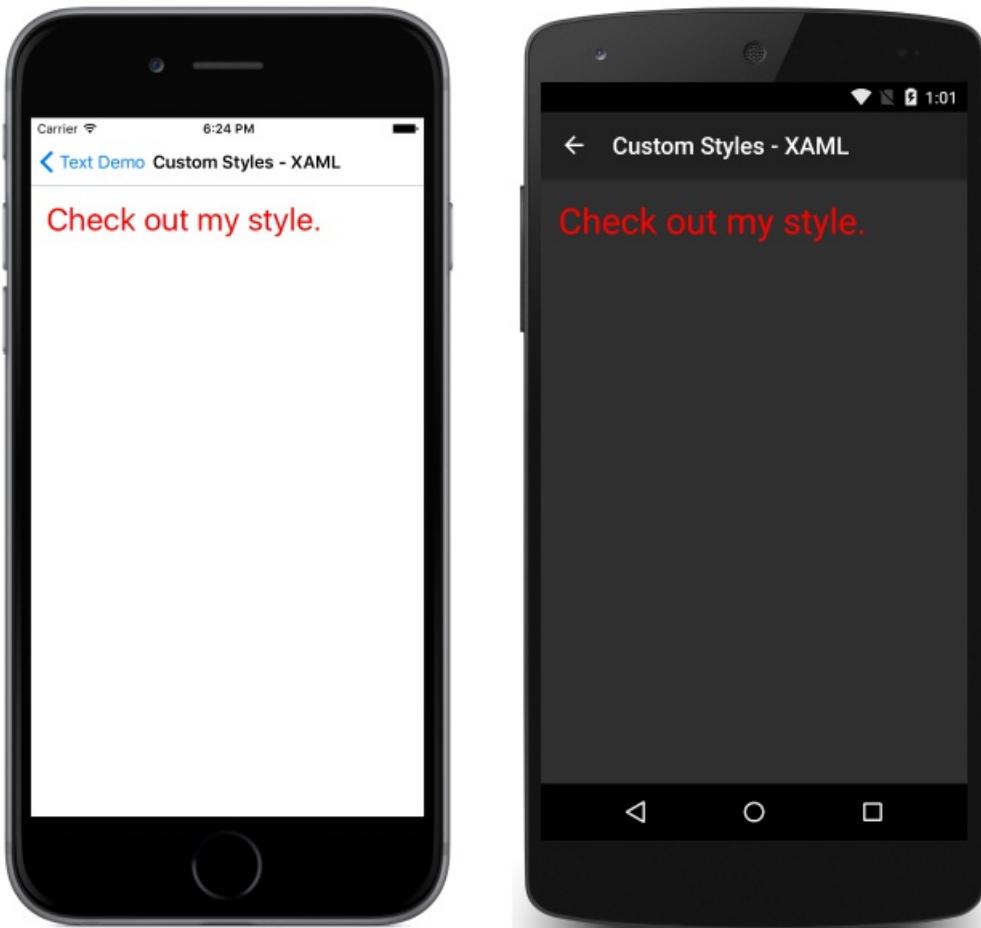
var label = new Label { Text = "Check out my style.", Style = LabelStyle };
```

In XAML:

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Style x:Key="LabelStyle" TargetType="Label">
            <Setter Property="TextColor" Value="Red"/>
            <Setter Property="FontSize" Value="30"/>
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<ContentPage.Content>
    <StackLayout>
        <Label Text="Check out my style." Style="{StaticResource LabelStyle}" />
    </StackLayout>
</ContentPage.Content>
```

Note that resources (including all styles) are defined within `ContentPage.Resources`, which is a sibling of the more familiar `ContentPage.Content` element.



## Applying Styles

Once a style has been created, it can be applied to any view matching its `TargetType`.

In XAML, custom styles are applied to views by supplying their `Style` property with a `StaticResource` markup extension referencing the desired style:

```
<Label Text="Check out my style." Style="{StaticResource LabelStyle}" />
```

In C#, styles can either be applied directly to a view or added to and retrieved from a page's `ResourceDictionary`.

To add directly:

```
var label = new Label { Text = "Check out my style.", Style = LabelStyle };
```

To add and retrieve from the page's `ResourceDictionary`:

```
this.Resources.Add ("LabelStyle", LabelStyle);
label.Style = (Style)Resources["LabelStyle"];
```

Built-in styles are applied differently, because they need to respond to accessibility settings. To apply built-in styles in XAML, the `DynamicResource` markup extension is used:

```
<Label Text="I'm a Title" Style="{DynamicResource TitleStyle}" />
```

In C#, built-in styles are selected from `Device.Styles`:

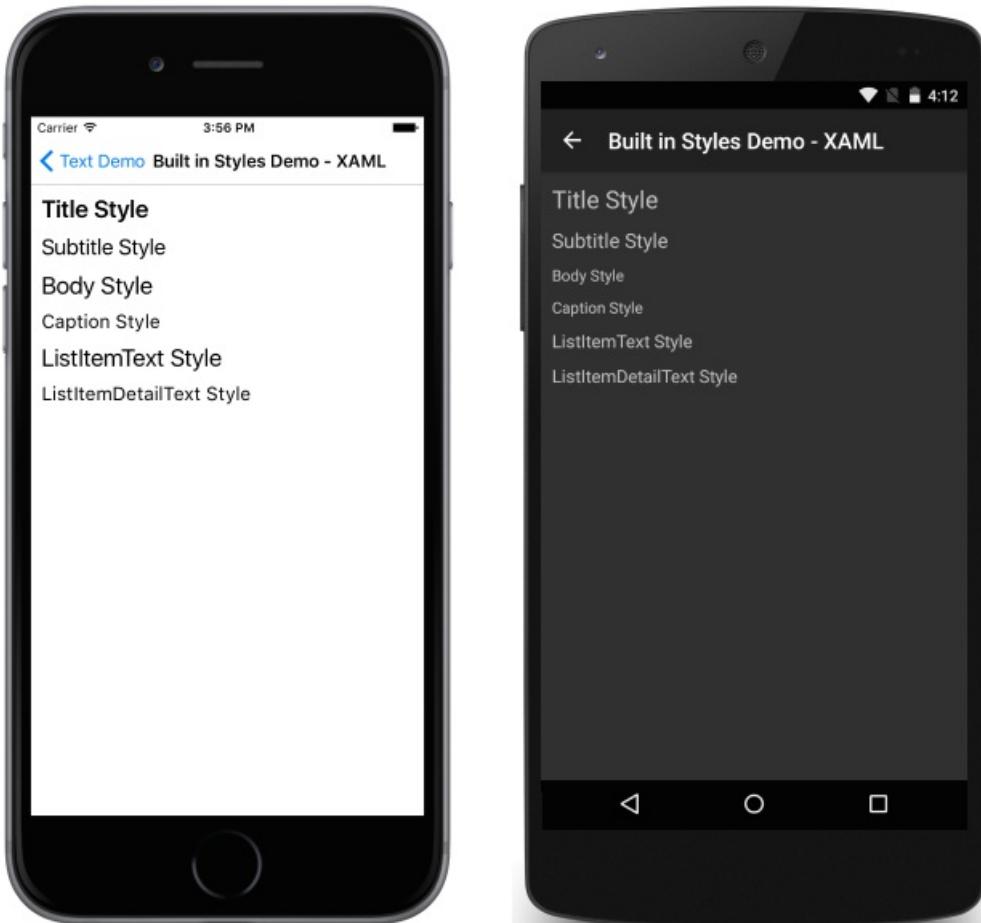
```
label.Style = Device.Styles.TitleStyle;
```

## Accessibility

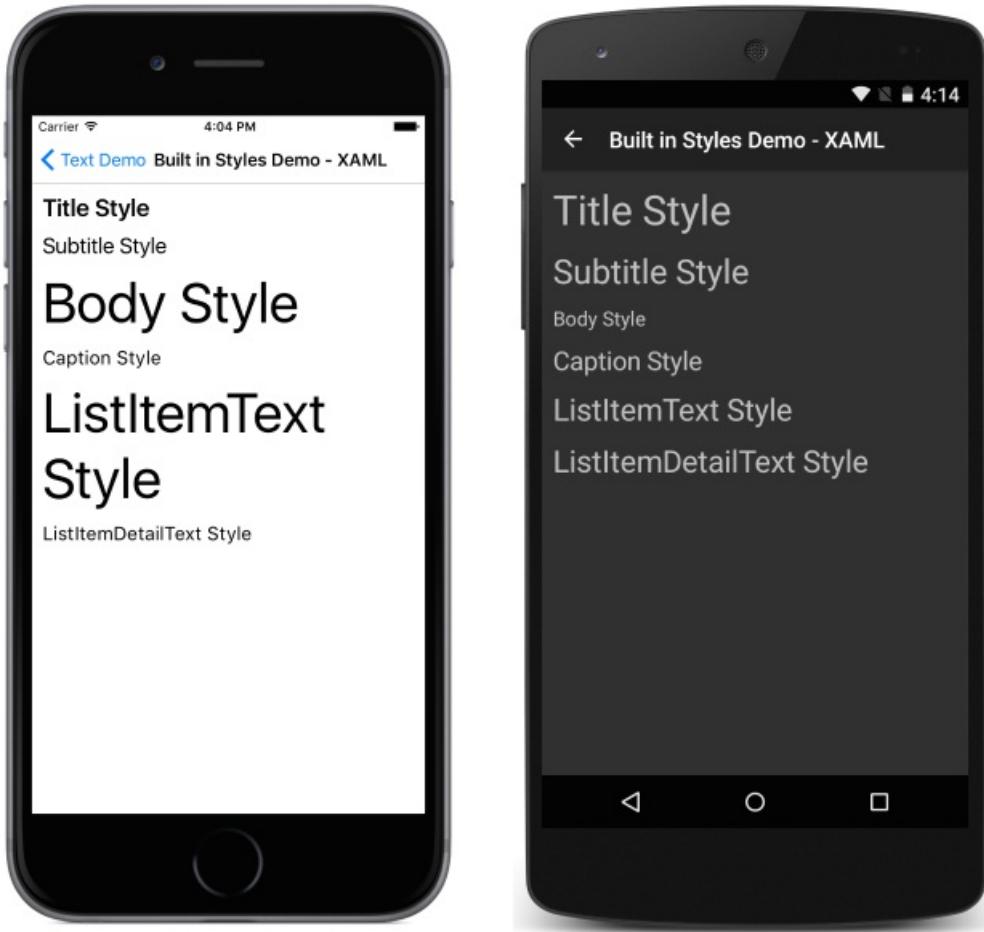
The built-in styles exist to make it easier to respect accessibility preferences. When using any of the built-in styles, font sizes will automatically increase if a user sets their accessibility preferences accordingly.

Consider the following example of the same page of views styled with the built-in styles with accessibility settings enabled and disabled:

Disabled:



Enabled:



To ensure accessibility, make sure that built-in styles are used as the basis for any text-related styles within your app, and that you are using styles consistently. See [Styles](#) for more details on extending and working with styles in general.

## Related Links

- [Creating Mobile Apps with Xamarin.Forms, Chapter 12](#)
- [Styles](#)
- [Text \(sample\)](#)
- [Style](#)

# Xamarin.Forms Themes

6/8/2018 • 2 minutes to read • [Edit Online](#)



Xamarin.Forms Themes were announced at Evolve 2016 and are available as a preview for customers to try and provide feedback.

A theme is added to a Xamarin.Forms application by including the **Xamarin.Forms.Theme.Base** Nuget package, plus an additional package that defines a specific theme (eg. Xamarin.Forms.Theme.Light) or else a local theme can be defined for the application.

Refer to the [Light theme](#) and [Dark theme](#) pages for instructions on how to add them to an app, or check out the [example custom theme](#).

**IMPORTANT:** You should also follow the steps to [load theme assemblies \(below\)](#) by adding some boilerplate code to the iOS `AppDelegate` and Android `MainActivity`. This will be improved in a future preview release.

## Control Appearance

The [Light](#) and [Dark](#) themes both define a specific visual appearance for the standard controls. Once you add a theme to the application's resource dictionary, the appearance of the standard controls will change.

The following XAML markup shows some common controls:

```
<StackLayout Padding="40">
    <Label Text="Regular label" />
    <Entry Placeholder="type here" />
    <Button Text="OK" />
    <BoxView Color="Yellow" />
    <Switch />
</StackLayout>
```

These screenshots show these controls with:

- No theme applied
- Light theme (only subtle differences to having no theme)
- Dark theme



## StyleClass

The `StyleClass` property allows a view's appearance to be changed according to a definition provided by a theme.

The [Light](#) and [Dark](#) themes both define three different appearances for a `BoxView`: `HorizontalRule`, `Circle`, and `Rounded`. This markup shows three different `BoxView`s with different style classes applied:

```
<StackLayout Padding="40">
    <BoxView StyleClass="HorizontalRule" />
    <BoxView StyleClass="Circle" />
    <BoxView StyleClass="Rounded" />
</StackLayout>
```

This renders with light and dark as follows:



## Built-in Classes

In addition to automatically styling the common controls the Light and Dark themes currently support the following classes that can be applied by setting the `StyleClass` on these controls:

### **BoxView**

- `HorizontalRule`
- `Circle`
- `Rounded`

### **Image**

- `Circle`
- `Rounded`
- `Thumbnail`

### **Button**

- `Default`
- `Primary`
- `Success`
- `Info`
- `Warning`
- `Danger`
- `Link`
- `Small`
- `Large`

### **Label**

- `Header`
- `Subheader`
- `Body`
- `Link`
- `Inverse`

# Troubleshooting

## Could not load file or assembly 'Xamarin.Forms.Theme.Light' or one of its dependencies

In the preview release, themes may not be able to load at runtime. Add the code shown below into the relevant projects to fix this error.

### iOS

In the **AppDelegate.cs** add the following lines after `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.iOS.UnderlineEffect);
```

### Android

In the **MainActivity.cs** add the following lines after `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.Android.UnderlineEffect);
```

## Related Links

- [ThemesDemo sample](#)

# Xamarin.Forms Light Theme

6/8/2018 • 2 minutes to read • [Edit Online](#)



## NOTE

Themes require the Xamarin.Forms 2.3 preview release. Check the [troubleshooting tips](#) if errors occur.

To use the Light Theme:

## 1. Add Nuget packages

- Xamarin.Forms.Theme.Base
- Xamarin.Forms.Theme.Light

## 2. Add to the Resource Dictionary

In the **App.xaml** file add a new custom `xmlns` for the theme, and then ensure the theme's resources are merged with the application's resource dictionary. An example XAML file is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="EvolveApp.App"
    xmlns:light="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Light">
    <Application.Resources>
        <ResourceDictionary MergedWith="light:LightThemeResources" />
    </Application.Resources>
</Application>
```

## 3. Load theme classes

Follow this [troubleshooting step](#) and add the required code in the iOS and Android application projects.

## 4. Use StyleClass

Here is an example of buttons and labels in the light theme, along with the markup that produces them.

[Light Theme](#) [Dark Theme >](#)[Button Class Success](#)[Button Class Info](#)[Button Class Warning](#)[Button Class Danger](#)[Button Class Link](#)[Button Class Default Small](#)[Button Class Default Large](#)

## Label Classes

[Label Class Header](#)[Label Class Subheader](#)[Label Class Body](#)

Speakers



Detail



Mini Hacks



Everything

```
<StackLayout Padding="20">
    <Button Text="Button Default" />
    <Button Text="Button Class Default" StyleClass="Default" />
    <Button Text="Button Class Primary" StyleClass="Primary" />
    <Button Text="Button Class Success" StyleClass="Success" />
    <Button Text="Button Class Info" StyleClass="Info" />
    <Button Text="Button Class Warning" StyleClass="Warning" />
    <Button Text="Button Class Danger" StyleClass="Danger" />
    <Button Text="Button Class Link" StyleClass="Link" />
    <Button Text="Button Class Default Small" StyleClass="Small" />
    <Button Text="Button Class Default Large" StyleClass="Large" />
</StackLayout>
```

The [complete list of built-in classes](#) shows what styles are available for some common controls.

# Xamarin.Forms Dark Theme

6/8/2018 • 2 minutes to read • [Edit Online](#)



## NOTE

Themes require the Xamarin.Forms 2.3 preview release. Check the [troubleshooting tips](#) if errors occur.

To use the Dark Theme:

## 1. Add Nuget packages

- Xamarin.Forms.Theme.Base
- Xamarin.Forms.Theme.Dark

## 2. Add to the Resource Dictionary

In the **App.xaml** file add a new custom `xmlns` for the theme, and then ensure the theme's resources are merged with the application's resource dictionary. An example XAML file is shown below:

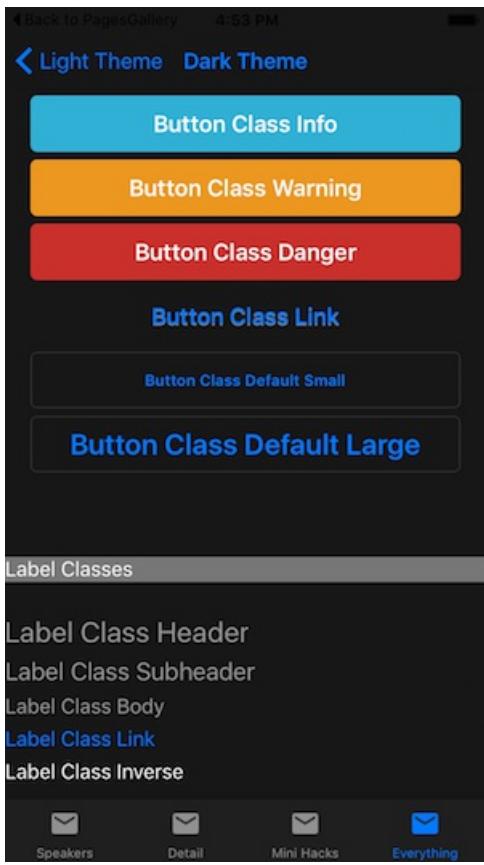
```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="EvolveApp.App"
    xmlns:dark="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Dark">
    <Application.Resources>
        <ResourceDictionary MergedWith="dark:DarkThemeResources" />
    </Application.Resources>
</Application>
```

## 3. Load theme classes

Follow this [troubleshooting step](#) and add the required code in the iOS and Android application projects.

## 4. Use StyleClass

Here is an example of buttons and labels in the dark theme, along with the markup that produces them.



```
<StackLayout Padding="20">
    <Button Text="Button Default" />
    <Button Text="Button Class Default" StyleClass="Default" />
    <Button Text="Button Class Primary" StyleClass="Primary" />
    <Button Text="Button Class Success" StyleClass="Success" />
    <Button Text="Button Class Info" StyleClass="Info" />
    <Button Text="Button Class Warning" StyleClass="Warning" />
    <Button Text="Button Class Danger" StyleClass="Danger" />
    <Button Text="Button Class Link" StyleClass="Link" />

    <Button Text="Button Class Default Small" StyleClass="Small" />
    <Button Text="Button Class Default Large" StyleClass="Large" />
</StackLayout>
```

The [complete list of built-in classes](#) shows what styles are available for some common controls.

# Creating a Custom Xamarin.Forms Theme

10/31/2018 • 3 minutes to read • [Edit Online](#)



In addition to adding a theme from a Nuget package (such as the [Light](#) and [Dark](#) themes), you can create your own resource dictionary themes that can be referenced in your app.

## Example

The three `BoxView`s shown on the [Themes page](#) are styled according to three classes defined in the two downloadable themes.

To understand how these work, the following markup creates an equivalent style that you could add directly to your `App.xaml`.

Note the `Class` attribute for `Style` (as opposed to the `x:Key` attribute available in earlier versions of Xamarin.Forms).

```
<ResourceDictionary>
    <!-- DEFINE ANY CONSTANTS -->
    <Color x:Key="SeparatorLineColor">#CCCCCC</Color>
    <Color x:Key="iOSDefaultTintColor">#007aff</Color>
    <Color x:Key="AndroidDefaultAccentColorColor">#1FAECE</Color>
    <OnPlatform x:TypeArguments="Color" x:Key="AccentColor">
        <On Platform="iOS" Value="{StaticResource iOSDefaultTintColor}" />
        <On Platform="Android" Value="{StaticResource AndroidDefaultAccentColorColor}" />
    </OnPlatform>
    <!-- BOXVIEW CLASSES -->
    <Style TargetType="BoxView" Class="HorizontalRule">
        <Setter Property="BackgroundColor" Value="{ StaticResource SeparatorLineColor }" />
        <Setter Property="HeightRequest" Value="1" />
    </Style>

    <Style TargetType="BoxView" Class="Circle">
        <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />
        <Setter Property="WidthRequest" Value="34"/>
        <Setter Property="HeightRequest" Value="34"/>
        <Setter Property="HorizontalOptions" Value="Start" />

        <Setter Property="local:ThemeEffects.Circle" Value="True" />
    </Style>

    <Style TargetType="BoxView" Class="Rounded">
        <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />
        <Setter Property="HorizontalOptions" Value="Start" />
        <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />

        <Setter Property="local:ThemeEffects.CornerRadius" Value="4" />
    </Style>
</ResourceDictionary>
```

You'll notice that the `Rounded` class refers to a custom effect `CornerRadius`. The code for this effect is given below - to reference it correctly a custom `xmlns` must be added to the `App.xaml`'s root element:

```
xmlns:local="clr-namespace:ThemesDemo;assembly=ThemesDemo"
```

## C# code in the .NET Standard library project or Shared Project

The code for creating a round-corner `BoxView` uses `effects`. The corner radius is applied using a `BindableProperty` and is implemented by applying an `effect`. The effect requires platform-specific code in the `iOS` and `Android` projects (shown below.)

```
namespace ThemesDemo
{
    public static class ThemeEffects
    {
        public static readonly BindableProperty CornerRadiusProperty =
            BindableProperty.CreateAttached("CornerRadius", typeof(double), typeof(ThemeEffects), 0.0,
            propertyChanged: OnChanged<CornerRadiusEffect, double>);

        private static void OnChanged<TEffect, TProp>(BindableObject bindable, object oldValue, object newValue)
            where TEffect : Effect, new()
        {
            if (!(bindable is View view))
            {
                return;
            }

            if (EqualityComparer<TProp>.Equals(newValue, default(TProp)))
            {
                var toRemove = view.Effects.FirstOrDefault(e => e is TEffect);
                if (toRemove != null)
                {
                    view.Effects.Remove(toRemove);
                }
            }
            else
            {
                view.Effects.Add(new TEffect());
            }
        }

        public static void SetCornerRadius(BindableObject view, double radius)
        {
            view.SetValue(CornerRadiusProperty, radius);
        }

        public static double GetCornerRadius(BindableObject view)
        {
            return (double)view.GetValue(CornerRadiusProperty);
        }

        private class CornerRadiusEffect : RoutingEffect
        {
            public CornerRadiusEffect()
                : base("Xamarin.CornerRadiusEffect")
            {
            }
        }
    }
}
```

## C# code in the iOS project

```
using System;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;
using CoreGraphics;
using Foundation;
using XFThemes;

namespace ThemesDemo.iOS
{
    public class CornerRadiusEffect : PlatformEffect
    {
        private nfloat _originalRadius;

        protected override void OnAttached()
        {
            if (Container != null)
            {
                _originalRadius = Container.Layer.CornerRadius;
                Container.ClipsToBounds = true;

                UpdateCorner();
            }
        }

        protected override void OnDetached()
        {
            if (Container != null)
            {
                Container.Layer.CornerRadius = _originalRadius;
                Container.ClipsToBounds = false;
            }
        }

        protected override void OnElementPropertyChanged(System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == ThemeEffects.CornerRadiusProperty.PropertyName)
            {
                UpdateCorner();
            }
        }

        private void UpdateCorner()
        {
            Container.Layer.CornerRadius = (nfloat)ThemeEffects.GetCornerRadius(Element);
        }
    }
}
```

### C# code in the Android project

```
using System;
using Xamarin.Forms.Platform;
using Xamarin.Forms.Platform.Android;
using Android.Views;
using Android.Graphics;

namespace ThemesDemo.Droid
{
    public class CornerRadiusEffect : BaseEffect
    {
        private ViewOutlineProvider _originalProvider;

        protected override bool CanBeApplied()
        {
            return Container != null && Android.OS.Build.VERSION.SdkInt >=
Android.OS.BuildVersionCodes.Lollipop;
        }

        protected override void OnAttachedInternal()
        {
            _originalProvider = Container.OutlineProvider;
            Container.OutlineProvider = new CornerRadiusOutlineProvider(Element);
            Container.ClipToOutline = true;
        }

        protected override void OnDetachedInternal()
        {
            Container.OutlineProvider = _originalProvider;
            Container.ClipToOutline = false;
        }

        protected override void OnElementPropertyChanged(System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (!Attached)
            {
                return;
            }

            if (args.PropertyName == ThemeEffects.CornerRadiusProperty.PropertyName)
            {
                Container.Invalidate();
            }
        }

        private class CornerRadiusOutlineProvider : ViewOutlineProvider
        {
            private Xamarin.Forms.Element _element;

            public CornerRadiusOutlineProvider(Xamarin.Forms.Element element)
            {
                _element = element;
            }

            public override void GetOutline(Android.Views.View view, Outline outline)
            {
                var pixels =
                    (float)ThemeEffects.GetCornerRadius(_element) *
                    view.Resources.DisplayMetrics.Density;

                outline.SetRoundRect(new Rect(0, 0, view.Width, view.Height), (int)pixels);
            }
        }
    }
}
```

## Summary

A custom theme can be created by defining styles for each control that requires custom appearance. Multiple styles for a control should be distinguished by different `Class` attributes in the resource dictionary, and then applied by setting the `StyleClass` attribute on the control.

A style can also utilize [effects](#) to further customize the appearance of a control.

[Implicit Styles](#) (without either a `x:Key` or `Style` attribute) continue to be applied to all controls that match the `TargetType`.

# Xamarin.Forms TimePicker

11/20/2018 • 5 minutes to read • [Edit Online](#)

A *Xamarin.Forms* view that allows the user to select a time.

The *Xamarin.Forms* `TimePicker` invokes the platform's time-picker control and allows the user to select a time. `TimePicker` defines the following properties:

- `Time` of type  `TimeSpan`, the selected time, which defaults to a  `TimeSpan` of 0. The  `TimeSpan` type indicates a duration of time since midnight.
- `Format` of type  `string`, a *standard* or *custom* .NET formatting string, which defaults to "t", the short time pattern.
- `TextColor` of type  `Color`, the color used to display the selected time, which defaults to `Color.Default`.
- `FontAttributes` of type  `FontAttributes`, which defaults to `FontAttributes.None`.
- `FontFamily` of type  `string`, which defaults to `null`.
- `FontSize` of type  `double`, which defaults to -1.0.

All of these properties are backed by `BindableProperty` objects, which means that they can be styled, and the properties can be targets of data bindings. The `Time` property has a default binding mode of `BindingMode.TwoWay`, which means that it can be a target of a data binding in an application that uses the *Model-View-ViewModel (MVVM)* architecture.

The `TimePicker` doesn't include an event to indicate a new selected `Time` value. If you need to be notified of this, you can add a handler for the `PropertyChanged` event.

## Initializing the Time property

In code, you can initialize the `Time` property to a value of type  `TimeSpan`:

```
TimePicker timePicker = new TimePicker
{
    Time = new TimeSpan(4, 15, 26) // Time set to "04:15:26"
};
```

When the `Time` property is specified in XAML, the value is converted to a  `TimeSpan` and validated to ensure that the number of milliseconds is greater than or equal to 0, and that the number of hours is less than 24. The time components should be separated by colons:

```
<TimePicker Time="4:15:26" />
```

If the `BindingContext` property of `TimePicker` is set to an instance of a ViewModel containing a property of type  `TimeSpan` named `SelectedTime` (for example), you can instantiate the `TimePicker` like this:

```
<TimePicker Time="{Binding SelectedTime}" />
```

In this example, the `Time` property is initialized to the `SelectedTime` property in the ViewModel. Because the `Time` property has a binding mode of `TwoWay`, any new time that the user selects is automatically propagated to the ViewModel.

If the `TimePicker` does not contain a binding on its `Time` property, an application should attach a handler to the `PropertyChanged` event to be informed when the user selects a new time.

For information about setting font properties, see [Fonts](#).

## TimePicker and layout

It's possible to use an unconstrained horizontal layout option such as `Center`, `Start`, or `End` with `TimePicker`:

```
<TimePicker ...
    HorizontalOptions="Center"
    ... />
```

However, this is not recommended. Depending on the setting of the `Format` property, selected times might require different display widths. For example, the "T" format string causes the `TimePicker` view to display times in a long format, and "4:15:26 AM" requires a greater display width than the short time format ("t") of "4:15 AM". Depending on the platform, this difference might cause the `TimePicker` view to change width in layout, or for the display to be truncated.

### TIP

It's best to use the default `HorizontalOptions` setting of `Fill` with `TimePicker`, and not to use a width of `Auto` when putting `TimePicker` in a `Grid` cell.

## TimePicker in an application

The [SetTimer](#) sample includes `TimePicker`, `Entry`, and `Switch` views on its page. The `TimePicker` can be used to select a time, and when that time occurs an alert dialog is displayed that reminds the user of the text in the `Entry`, provided the `Switch` is toggled on. Here's the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:SetTimer"
    x:Class="SetTimer.MainPage">
<StackLayout>
    ...
    <Entry x:Name="_entry"
        Placeholder="Enter event to be reminded of" />
    <Label Text="Select the time below to be reminded at." />
    <TimePicker x:Name="_timePicker"
        Time="11:00:00"
        Format="T"
        PropertyChanged="OnTimePickerPropertyChanged" />
    <StackLayout Orientation="Horizontal">
        <Label Text="Enable timer:" />
        <Switch x:Name="_switch"
            HorizontalOptions="EndAndExpand"
            Toggled="OnSwitchToggled" />
    </StackLayout>
</StackLayout>
</ContentPage>
```

The `Entry` lets you enter reminder text that will be displayed when the selected time occurs. The `TimePicker` is assigned a `Format` property of "T" for long time format. It has an event handler attached to the `PropertyChanged` event, and the `Switch` has a handler attached to its `Toggled` event. These events handlers are in the code-behind file and call the `setTriggerTime` method:

```

public partial class MainPage : ContentPage
{
    DateTime _triggerTime;

    public MainPage()
    {
        InitializeComponent();

        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerTick);
    }

    bool OnTimerTick()
    {
        if (_switch.IsToggled && DateTime.Now >= _triggerTime)
        {
            _switch.IsToggled = false;
            DisplayAlert("Timer Alert", "The '" + _entry.Text + "' timer has elapsed", "OK");
        }
        return true;
    }

    void OnTimePickerPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        if (args.PropertyName == "Time")
        {
            SetTriggerTime();
        }
    }

    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        SetTriggerTime();
    }

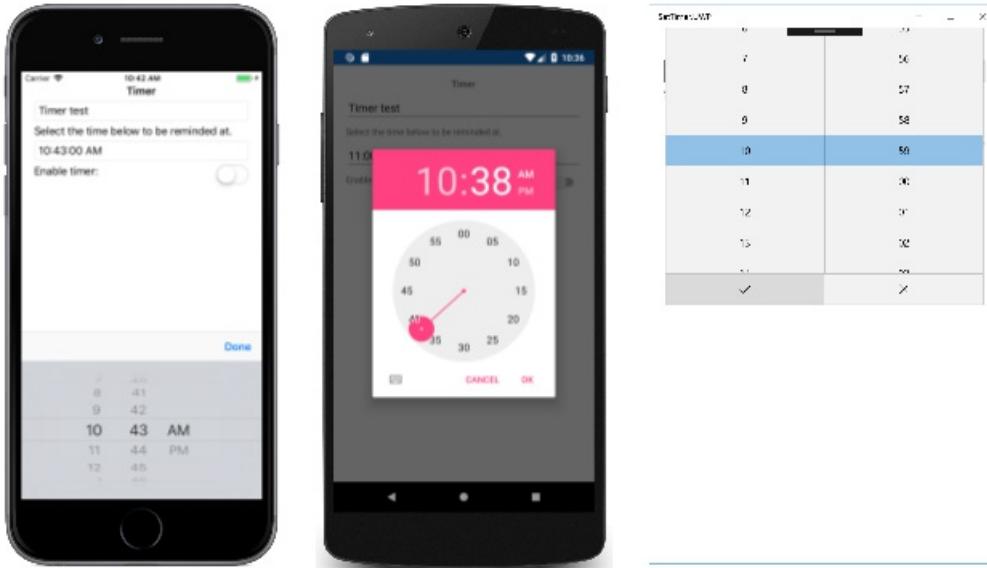
    void SetTriggerTime()
    {
        if (_switch.IsToggled)
        {
            _triggerTime = DateTime.Today + _timePicker.Time;
            if (_triggerTime < DateTime.Now)
            {
                _triggerTime += TimeSpan.FromDays(1);
            }
        }
    }
}

```

The `SetTriggerTime` method calculates a timer time based on the `DateTime.Today` property value and the `TimeSpan` value returned from the `TimePicker`. This is necessary because the `DateTime.Today` property returns a `DateTime` indicating the current date, but with a time of midnight. If the timer time has already passed today, then it's assumed to be tomorrow.

The timer ticks every second, executing the `OnTimerTick` method that checks whether the `Switch` is on and whether the current time is greater than or equal to the timer time. When the timer time occurs, the `DisplayAlert` method presents an alert dialog to the user as a reminder.

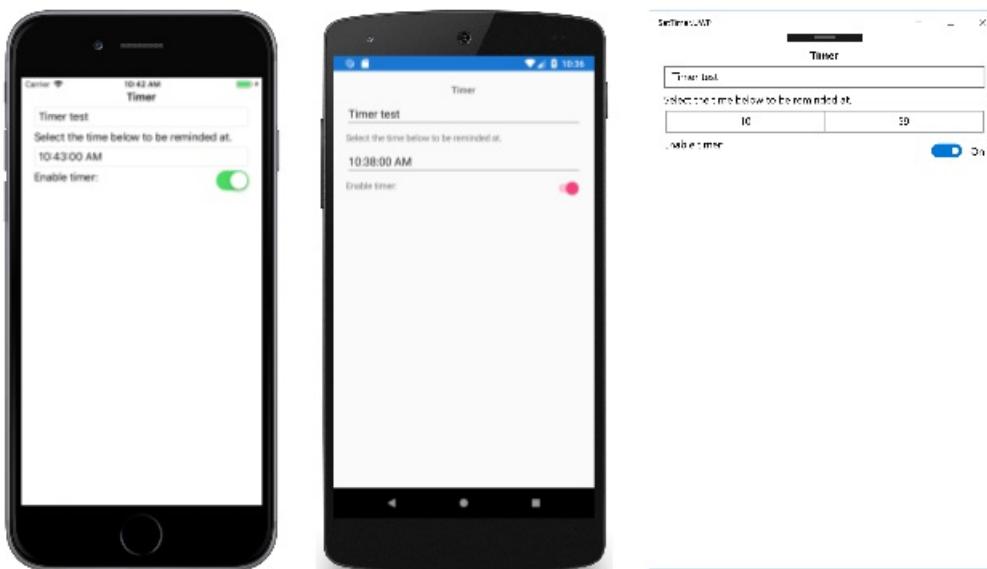
When the sample is first run, the `TimePicker` view is initialized to 11am. Tapping the `TimePicker` invokes the platform time picker. The platforms implement the time picker in very different ways, but each approach is familiar to users of that platform:



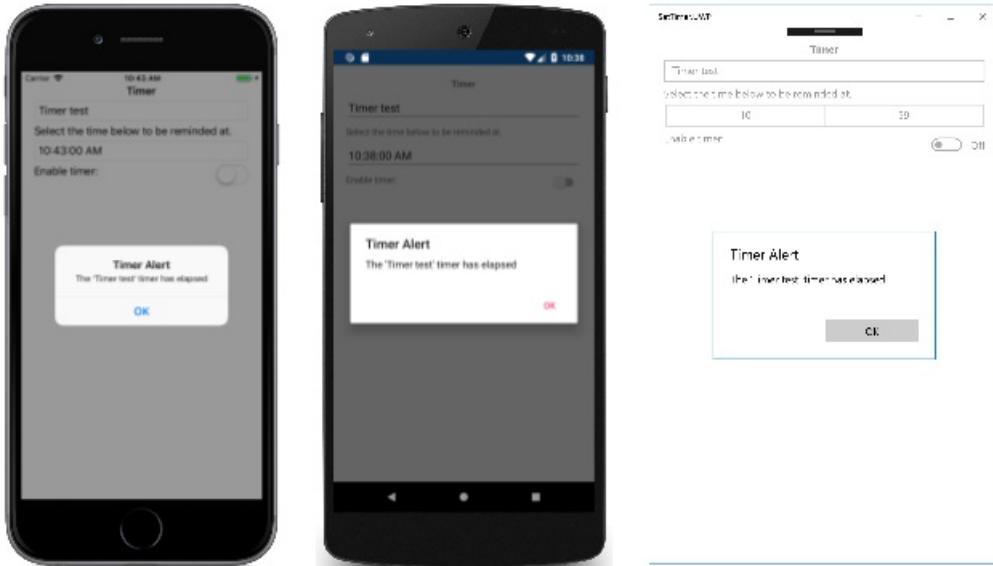
### TIP

On Android, the `TimePicker` dialog can be customized by overriding the `CreateTimePickerDialog` method in a custom renderer. This allows, for example, additional buttons to be added to the dialog.

After selecting a time, the selected time is displayed in the `TimePicker`:



Provided that the `Switch` is toggled to the on position, the application displays an alert dialog reminding the user of the text in the `Entry` when the selected time occurs:



As soon as the alert dialog is displayed, the [Switch](#) is toggled to the off position.

## Related links

- [SetTimer sample](#)
- [TimePicker API](#)

# The Xamarin.Forms Visual State Manager

11/12/2018 • 15 minutes to read • [Edit Online](#)

Use the Visual State Manager to make changes to XAML elements based on visual states set from code.

The Visual State Manager (VSM) is new in Xamarin.Forms 3.0. The VSM provides a structured way to make visual changes to the user interface from code. In most cases, the user interface of the application is defined in XAML, and this XAML includes markup describing how the Visual State Manager affects the visuals of the user interface.

The VSM introduces the concept of *visual states*. A Xamarin.Forms view such as a `Button` can have several different visual appearances depending on its underlying state — whether it's disabled, or pressed, or has input focus. These are the button's states.

Visual states are collected in *visual state groups*. All the visual states within a visual state group are mutually exclusive. Both visual states and visual state groups are identified by simple text strings.

The Xamarin.Forms Visual State Manager defines one visual state group named "CommonStates" with three visual states:

- "Normal"
- "Disabled"
- "Focused"

This visual state group is supported for all classes that derive from `VisualElement`, which is the base class for `View` and `Page`.

You can also define your own visual state groups and visual states, as this article will demonstrate.

## NOTE

Xamarin.Forms developers familiar with [triggers](#) are aware that triggers can also make changes to visuals in the user interface based on changes in a view's properties or the firing of events. However, using triggers to deal with various combinations of these changes can become quite confusing. Historically, the Visual State Manager was introduced in Windows XAML-based environments to alleviate the confusion resulting from combinations of visual states. With the VSM, the visual states within a visual state group are always mutually exclusive. At any time, only one state in each group is the current state.

## The common states

The Visual State Manager allows you to include sections in your XAML file that can change the visual appearance of a view if the view is normal, or disabled, or has the input focus. These are known as the *common states*.

For example, suppose you have an `Entry` view on your page, and you want the visual appearance of the `Entry` to change in the following ways:

- The `Entry` should have a pink background when the `Entry` is disabled.
- The `Entry` should have a lime background normally.
- The `Entry` should expand to twice its normal height when it has input focus.

You can attach the VSM markup to an individual view, or you can define it in a style if it applies to multiple views. The next two sections describe these approaches.

## VSM markup on a view

To attach VSM markup to an `Entry` view, first separate the `Entry` into start and end tags:

```
<Entry FontSize="18">  
  </Entry>
```

It's given an explicit font size because one of the states will use the `FontSize` property to double the size of the text in the `Entry`.

Next, insert `VisualStateManager.VisualStateGroups` tags between those tags:

```
<Entry FontSize="18">  
  <VisualStateManager.VisualStateGroups>  
  
  </VisualStateManager.VisualStateGroups>  
</Entry>
```

`VisualStateGroups` is an attached bindable property defined by the `VisualStateManager` class. (For more information on attached bindable properties, see the article [Attached properties](#).) This is how the `VisualStateGroups` property is attached to the `Entry` object.

The `visualStateGroups` property is of type `VisualStateGroupList`, which is a collection of `VisualStateGroup` objects. Within the `VisualStateManager.VisualStateGroups` tags, insert a pair of `VisualStateGroup` tags for each group of visual states you wish to include:

```
<Entry FontSize="18">  
  <VisualStateManager.VisualStateGroups>  
    <VisualStateGroup x:Name="CommonStates">  
  
      </VisualStateGroup>  
    </VisualStateManager.VisualStateGroups>  
</Entry>
```

Notice that the `VisualStateGroup` tag has an `x:Name` attribute indicating the name of the group. The `VisualStateGroup` class defines a `Name` property that you can use instead:

```
<VisualStateGroup Name="CommonStates">
```

You can use either `x:Name` or `Name` but not both in the same element.

The `visualStateGroup` class defines a property named `States`, which is a collection of `VisualState` objects. `States` is the *content property* of `visualStateGroups` so you can include the `VisualState` tags directly between the `VisualStateGroup` tags. (Content properties are discussed in the article [Essential XAML Syntax](#).)

The next step is to include a pair of tags for every visual state in that group. These also can be identified using `x:Name` OR `Name`:

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                </VisualState>

            <VisualState x:Name="Focused">
                </VisualState>

            <VisualState x:Name="Disabled">
                </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

`VisualState` defines a property named `Setters`, which is a collection of `Setter` objects. These are the same `Setter` objects that you use in a `Style` object.

`Setters` is *not* the content property of `VisualState`, so it is necessary to include property element tags for the `Setters` property:

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Focused">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Disabled">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

You can now insert one or more `Setter` objects between each pair of `Setters` tags. These are the `Setter` objects that define the visual states described earlier:

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Lime" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Focused">
                <VisualState.Setters>
                    <Setter Property="FontSize" Value="36" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Disabled">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Pink" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

Each `Setter` tag indicates the value of a particular property when that state is current. Any property referenced by a `setter` object must be backed by a bindable property.

Markup similar to this is the basis of the **VSM on View** page in the **VsmDemos** sample program. The page includes three `Entry` views, but only the second one has the VSM markup attached to it:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:VsmDemos"
    x:Class="VsmDemos.MainPage"
    Title="VSM Demos">

    <StackLayout>
        <StackLayout.Resources>
            <Style TargetType="Entry">
                <Setter Property="Margin" Value="20, 0" />
                <Setter Property="FontSize" Value="18" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="Margin" Value="20, 30, 20, 0" />
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </StackLayout.Resources>

        <Label Text="Normal Entry:" />

        <Entry />

        <Label Text="Entry with VSM: " />

        <Entry>
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CommonStates">

                    <VisualState x:Name="Normal">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor" Value="Lime" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState x:Name="Focused">
                        <VisualState.Setters>
                            <Setter Property="FontSize" Value="36" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState x:Name="Disabled">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor" Value="Pink" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

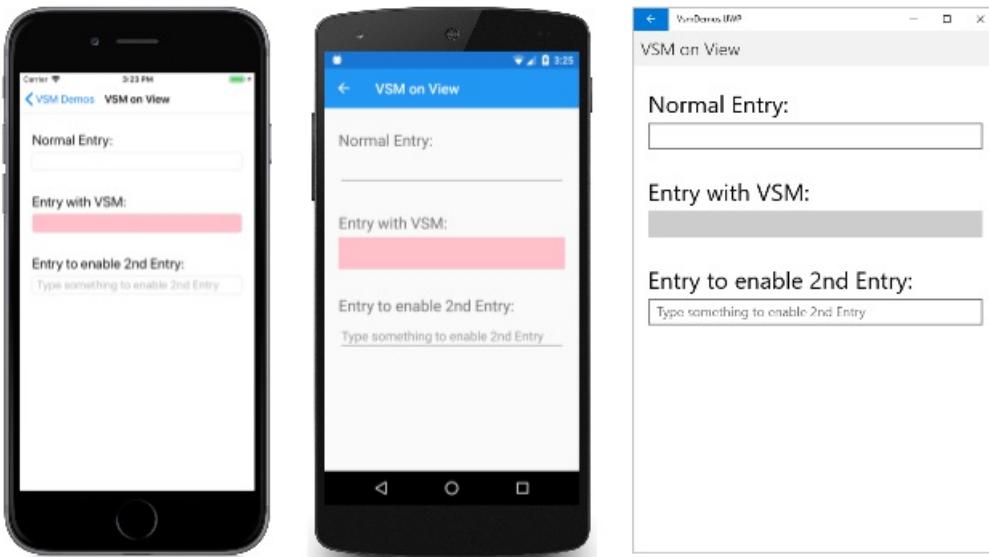
            <Entry.Triggers>
                <DataTrigger TargetType="Entry"
                    Binding="{Binding Source={x:Reference entry3},
                        Path=Text.Length}"
                    Value="0">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Entry.Triggers>
        </Entry>

        <Label Text="Entry to enable 2nd Entry:" />

        <Entry x:Name="entry3"
            Text=""
            Placeholder="Type something to enable 2nd Entry" />
    </StackLayout>
</ContentPage>

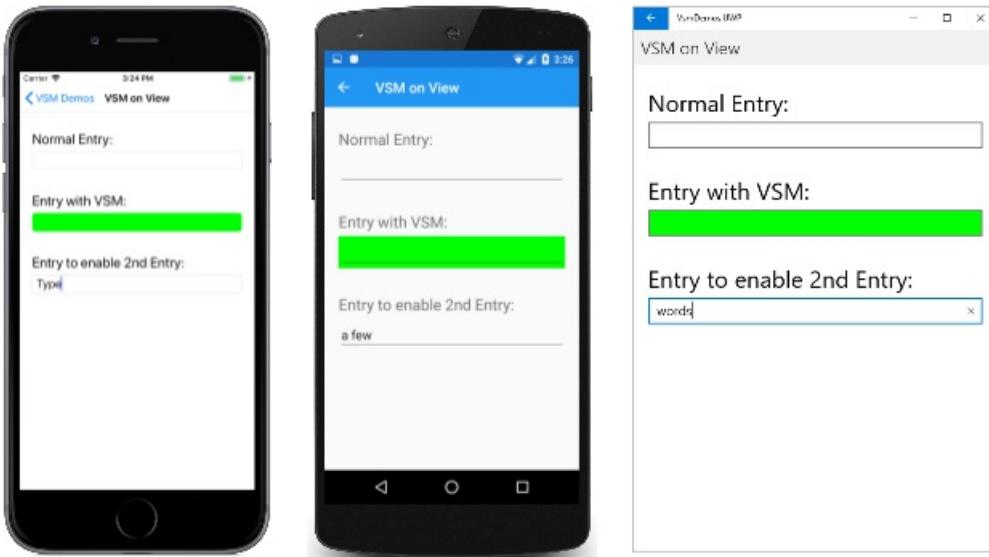
```

Notice that the second `Entry` also has a `DataTrigger` as part of its `Trigger` collection. This causes the `Entry` to be disabled until something is typed into the third `Entry`. Here's the page at startup running on iOS, Android, and the Universal Windows Platform (UWP):

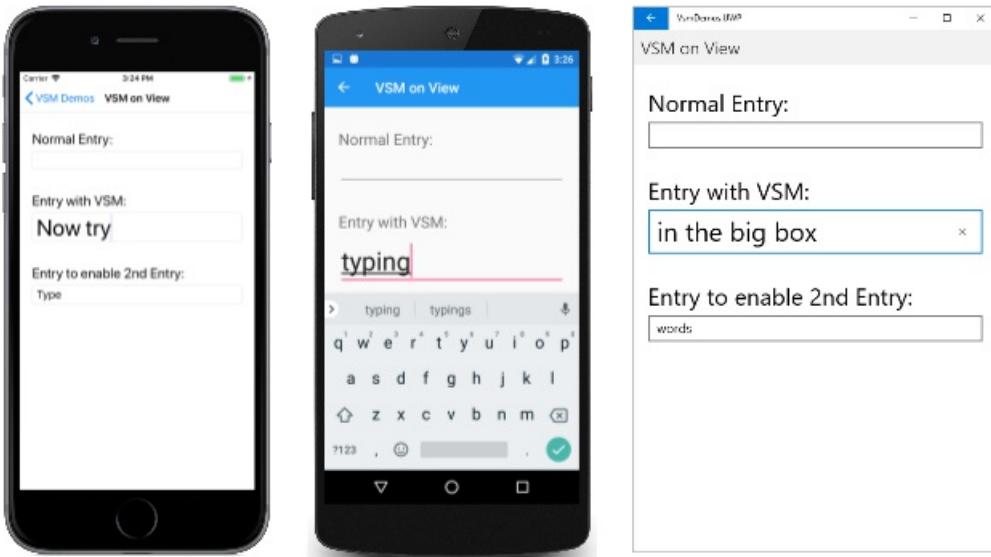


The current visual state is "Disabled" so the background of the second `Entry` is pink on the iOS and Android screens. The UWP implementation of `Entry` does not allow setting the background color when the `Entry` is disabled.

When you enter some text into the third `Entry`, the second `Entry` switches into the "Normal" state, and the background is now lime:



When you touch the second `Entry`, it gets the input focus. It switches to the "Focused" state and expands to twice its height:



Notice that the `Entry` does not retain the lime background when it gets the input focus. As the Visual State Manager switches between the visual states, the properties set by the previous state are unset. Keep in mind that the visual states are mutually exclusive. The "Normal" state does not mean solely that the `Entry` is enabled. It means that the `Entry` is enabled and does not have input focus.

If you want the `Entry` to have a lime background in the "Focused" state, add another `Setter` to that visual state:

```
<VisualState x:Name="Focused">
    <VisualState.Setters>
        <Setter Property="FontSize" Value="36" />
        <Setter Property="BackgroundColor" Value="Lime" />
    </VisualState.Setters>
</VisualState>
```

In order for these `Setter` objects to work properly, a `VisualStateGroup` must contain `VisualState` objects for all the states in that group. If there is a visual state that does not have any `Setter` objects, include it anyway as an empty tag:

```
<VisualState x:Name="Normal" />
```

### Visual State Manager markup in a style

It's often necessary to share the same Visual State Manager markup among two or more views. In this case, you'll want to put the markup in a `Style` definition.

Here's the existing implicit `Style` for the `Entry` elements in the **VSM On View** page:

```
<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
</Style>
```

Add `Setter` tags for the `VisualStateManager.VisualStateGroups` attached bindable property:

```

<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">
        </Setter>
    </Style>

```

The content property for `Setter` is `Value`, so the value of the `Value` property can be specified directly within those tags. That property is of type `VisualStateGroupList`:

```

<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            </VisualStateGroupList>
        </Setter>
    </Style>

```

Within those tags you can include one or more `VisualStateGroup` objects:

```

<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup x:Name="CommonStates">
                </VisualStateGroup>
            </VisualStateGroupList>
        </Setter>
    </Style>

```

The remainder of the VSM markup is the same as before.

Here's the **VSM in Style** page showing the complete VSM markup:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="VsmDemos.VsmInStylePage"
    Title="VSM in Style">
    <StackLayout>
        <StackLayout.Resources>
            <Style TargetType="Entry">
                <Setter Property="Margin" Value="20, 0" />
                <Setter Property="FontSize" Value="18" />
                <Setter Property="VisualStateManager.VisualStateGroups">
                    <VisualStateGroupList>
                        <VisualStateGroup x:Name="CommonStates">

                            <VisualState x:Name="Normal">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor" Value="Lime" />
                                </VisualState.Setters>
                            </VisualState>

                            <VisualState x:Name="Focused">
                                <VisualState.Setters>
                                    <Setter Property="FontSize" Value="36" />
                                    <Setter Property="BackgroundColor" Value="Lime" />
                                </VisualState.Setters>
                            </VisualState>

                            <VisualState x:Name="Disabled">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor" Value="Pink" />
                                </VisualState.Setters>
                            </VisualState>
                        </VisualStateGroup>
                    </VisualStateGroupList>
                </Setter>
            </Style>

            <Style TargetType="Label">
                <Setter Property="Margin" Value="20, 30, 20, 0" />
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </StackLayout.Resources>

        <Label Text="Normal Entry:" />

        <Entry />

        <Label Text="Entry with VSM: " />

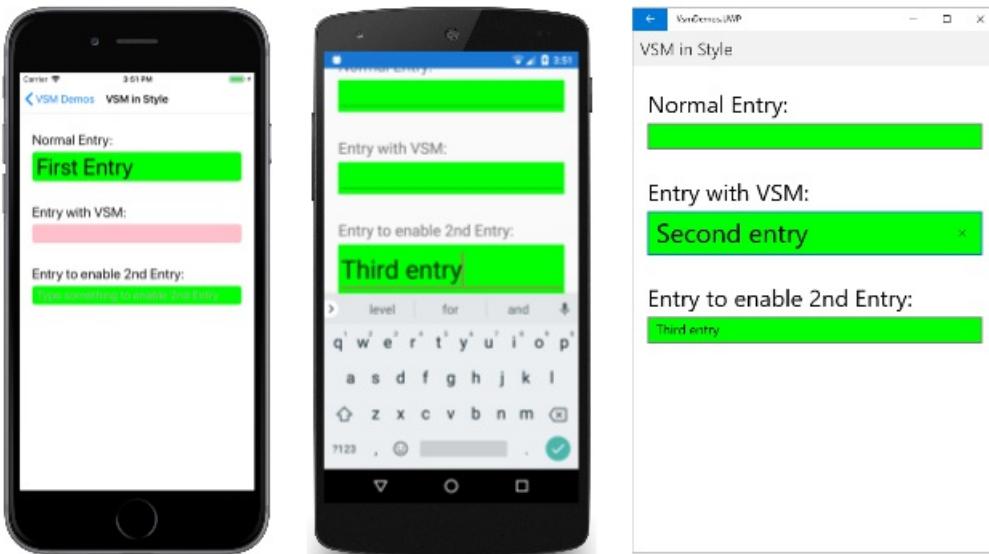
        <Entry>
            <Entry.Triggers>
                <DataTrigger TargetType="Entry"
                    Binding="{Binding Source={x:Reference entry3},
                        Path=Text.Length}"
                    Value="0">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Entry.Triggers>
        </Entry>

        <Label Text="Entry to enable 2nd Entry:" />

        <Entry x:Name="entry3"
            Text=""
            Placeholder="Type something to enable 2nd Entry" />
    </StackLayout>
</ContentPage>

```

Now all the `Entry` views on this page respond the same way to their visual states. Notice also that the "Focused" state now includes a second `Setter` that gives each `Entry` a lime background also when it has input focus:



## Defining your own visual states

Every class that derives from `VisualElement` supports the three common states "Normal", "Focused", and "Disabled". Internally, the `VisualElement` class detects when it's becoming enabled or disabled, or focused or unfocused, and calls the static `VisualStateManager.GoToState` method:

```
VisualStateManager.GoToState(this, "Focused");
```

This is the only Visual State Manager code that you'll find in the `visualElement` class. Because `GoToState` is called for every object based on every class that derives from `VisualElement`, you can use the Visual State Manager with any `VisualElement` object to respond to these changes.

Interestingly, the name of the visual state group "CommonStates" is not explicitly referenced in `visualElement`. The group name is not part of the API for the Visual State Manager. Within one of the two sample program shown so far, you can change the name of the group from "CommonStates" to anything else, and the program will still work. The group name is merely a general description of the states in that group. It is implicitly understood that the visual states in any group are mutually exclusive: One state and only one state is current at any time.

If you want to implement your own visual states, you'll need to call `VisualStateManager.GoToState` from code. Most often you'll make this call from the code-behind file of your page class.

The **VSM Validation** page in the **VsmDemos** sample shows how to use the Visual State Manager in connection with input validation. The XAML file consists of two `Label` elements, an `Entry`, and `Button`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="VsmDemos.VsmValidationPage"
    Title="VSM Validation">

    <StackLayout Padding="10, 10">

        <Label Text="Enter a U.S. phone number:"
            FontSize="Large" />

        <Entry Placeholder="555-555-5555"
            FontSize="Large"
            Margin="30, 0, 0, 0"
            TextChanged="OnTextChanged" />

        <Label x:Name="helpLabel"
            Text="Phone number must be of the form 555-555-5555, and not begin with a 0 or 1">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ValidityStates">

                    <VisualState Name="Valid">
                        <VisualState.Setters>
                            <Setter Property="TextColor" Value="Transparent" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="Invalid" />

                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Label>

        <Button x:Name="submitButton"
            Text="Submit"
            FontSize="Large"
            Margin="0, 20"
            VerticalOptions="Center"
            HorizontalOptions="Center">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ValidityStates">

                    <VisualState Name="Valid" />

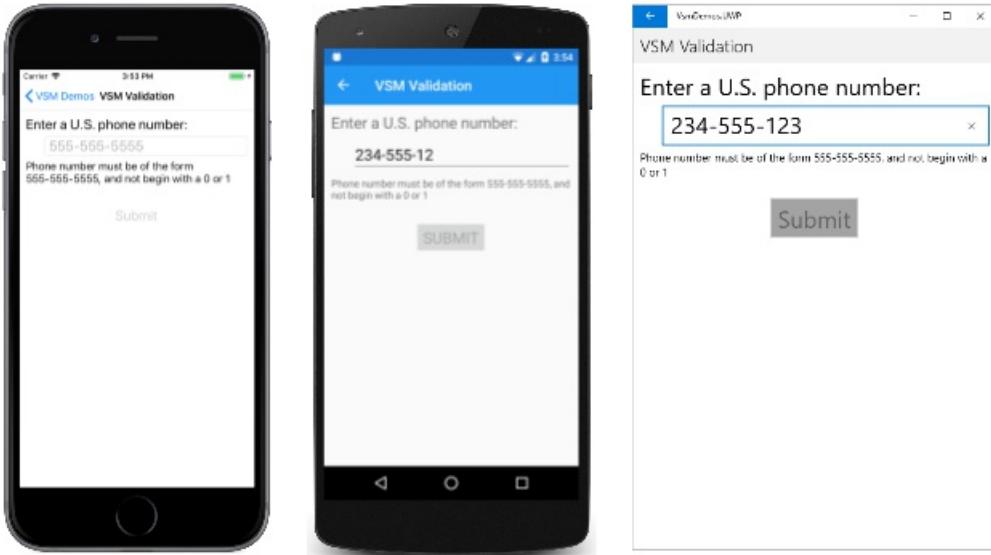
                    <VisualState Name="Invalid">
                        <VisualState.Setters>
                            <Setter Property="IsEnabled" Value="False" />
                        </VisualState.Setters>
                    </VisualState>

                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Button>
    </StackLayout>
</ContentPage>

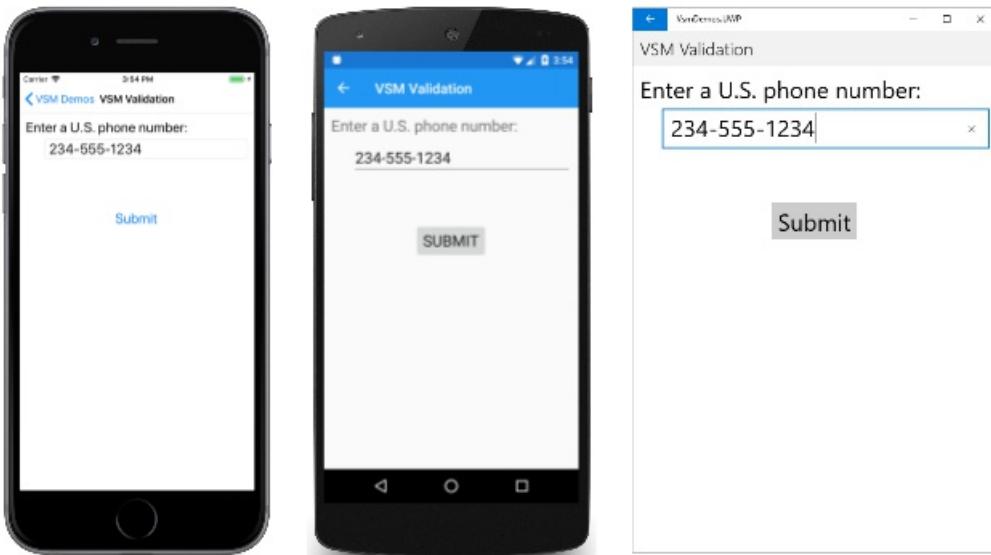
```

VSM markup is attached to the second `Label` (named `helpLabel`) and the `Button` (named `submitButton`). There are two mutually-exclusive states, named "Valid" and "Invalid". Notice that each of the two "ValidationState" groups contains `VisualState` tags for both "Valid" and "Invalid", although one of them is empty in each case.

If the `Entry` does not contain a valid phone number, then the current state is "Invalid", and so the second `Label` is visible and the `Button` is disabled:



When a valid phone number is entered, then the current state becomes "Valid". The second `Entry` disappears and the `Button` is now enabled:



The code-behind file is responsible for handling the `TextChanged` event from the `Entry`. The handler uses a regular expression to determine if the input string is valid or not. The method in the code-behind file named `GoToState` calls the static `VisualStateManager.GoToState` method for both `helpLabel` and `submitButton`:

```

public partial class VsmValidationPage : ContentPage
{
    public VsmValidationPage ()
    {
        InitializeComponent ();

        GoToState(false);
    }

    void OnTextChanged(object sender, TextChangedEventArgs args)
    {
        bool isValid = Regex.IsMatch(args.NewTextValue, @"^[2-9]\d{2}-\d{3}-\d{4}$");
        GoToState(isValid);
    }

    void GoToState(bool isValid)
    {
        string visualState = isValid ? "Valid" : "Invalid";
        VisualStateManager.GoToState(helpLabel, visualState);
        VisualStateManager.GoToState(submitButton, visualState);
    }
}

```

Notice also that the `GoToState` method is called from the constructor to initialize the state. There should always be a current state. But nowhere in the code is there any reference to the name of the visual state group, although it's referenced in the XAML as "ValidationStates" for purposes of clarity.

Notice that the code-behind file must take account of every object on the page that is affected by these visual states, and to call `VisualStateManager.GoToState` for each of these objects. In this example, it's only two objects (the `Label` and the `Button`), but it could be several more.

You might wonder: If the code-behind file must reference every object on the page that is affected by these visual states, why can't the code-behind file simply access the objects directly? It surely could. However, the advantage of using the VSM is that you can control how visual elements react to different state entirely in XAML, which keeps all of the UI design in one location. This avoids setting visual appearance by accessing visual elements directly from the code-behind.

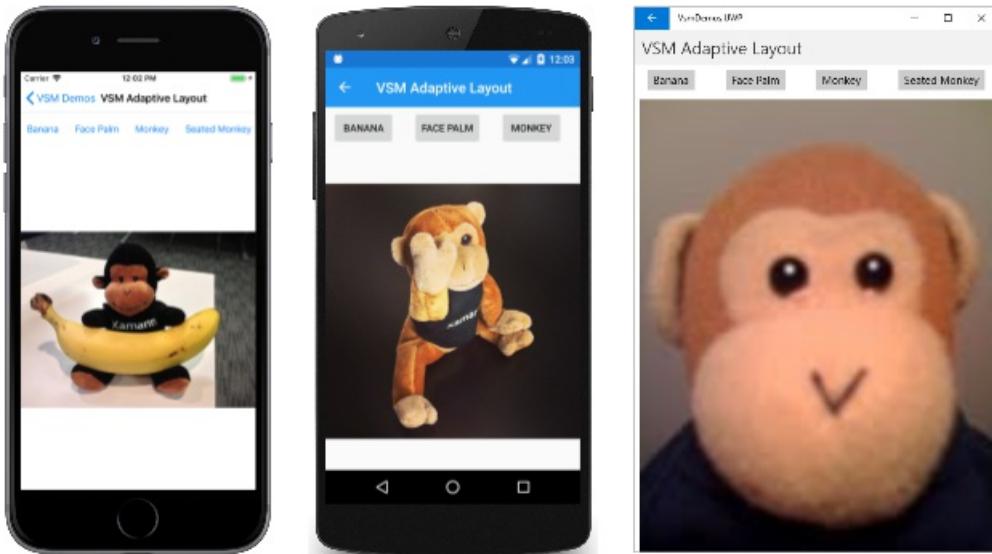
It might be tempting to consider deriving a class from `Entry` and perhaps defining a property that you can set to an external validation function. The class that derives from `Entry` can then call the `VisualStateManager.GoToState` method. This scheme would work fine, but only if the `Entry` were the only object affected by the different visual states. In this example, a `Label` and a `Button` are also be affected. There is no way for VSM markup attached to an `Entry` to control other objects on the page, and no way for VSM markup attached to these other objects to reference a change in visual state from another object.

## Using the Visual State Manager for adaptive layout

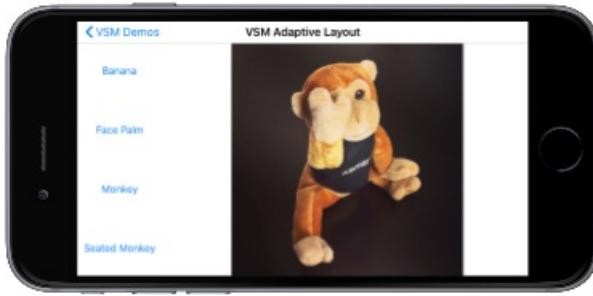
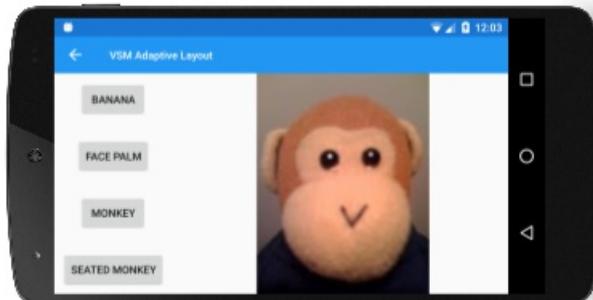
A Xamarin.Forms application running on a phone can usually be viewed in a portrait or landscape aspect ratio, and a Xamarin.Forms program running on the desktop can be resized to assume many different sizes and aspect ratios. A well-designed application might display its content differently for these various page or window form factors.

This technique is sometimes known as *adaptive layout*. Because adaptive layout solely involves a program's visuals, it is an ideal application of the Visual State Manager.

A simple example is an application that displays a small collection of buttons that affect the application's content. In portrait mode, these buttons might be displayed in a horizontal row on the top of the page:



In landscape mode, the array of buttons might be moved to one side, and displayed in a column:



From top to bottom, the program is running on the Universal Windows Platform, Android, and iOS.

The **VSM Adaptive Layout** page in the [VsmDemos](#) sample defines a group named "OrientationStates" with two visual states named "Portrait" and "Landscape". (A more complex approach might be based on several different page or window widths.)

VSM markup occurs in four places in the XAML file. The `StackLayout` named `mainStack` contains both the menu and the content, which is an `Image` element. This `StackLayout` should have a vertical orientation in portrait mode and a horizontal orientation in landscape mode:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="VsmDemos.VsmAdaptiveLayoutPage"
    Title="VSM Adaptive Layout">

    <StackLayout x:Name="mainStack">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="OrientationStates">

                <VisualState Name="Portrait">
                    <VisualState.Setters>
                        <Setter Property="Orientation" Value="Vertical" />
                    </VisualState.Setters>
                </VisualState>

                <VisualState Name="Landscape">
                    <VisualState.Setters>
                        <Setter Property="Orientation" Value="Horizontal" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>

        <ScrollView x:Name="menuScroll">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="OrientationStates">

                    <VisualState Name="Portrait">
                        <VisualState.Setters>
                            <Setter Property="Orientation" Value="Horizontal" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="Landscape">
                        <VisualState.Setters>
                            <Setter Property="Orientation" Value="Vertical" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

            <StackLayout x:Name="menuStack">
                <VisualStateManager.VisualStateGroups>
                    <VisualStateGroup Name="OrientationStates">

                        <VisualState Name="Portrait">
                            <VisualState.Setters>
                                <Setter Property="Orientation" Value="Horizontal" />
                            </VisualState.Setters>
                        </VisualState>

                        <VisualState Name="Landscape">
                            <VisualState.Setters>
                                <Setter Property="Orientation" Value="Vertical" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateManager.VisualStateGroups>

                <StackLayout.Resources>
                    <Style TargetType="Button">
                        <Setter Property="VisualStateManager.VisualStateGroups">
                            <VisualStateGroupList>
                                <VisualStateGroup Name="OrientationStates">

                                    <VisualState Name="Portrait">
                                        <VisualState.Setters>
                                            <Setter Property="HorizontalOptions" Value="CenterAndExpand" />
                                        </VisualState.Setters>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateGroupList>
                        </Setter>
                    </Style>
                </StackLayout.Resources>
            </StackLayout>
        </ScrollView>
    </StackLayout>

```

```

        <Setter Property="Margin" Value="10, 5" />
    </VisualState.Setters>
</VisualState>

<VisualState Name="Landscape">
    <VisualState.Setters>
        <Setter Property="VerticalOptions" Value="CenterAndExpand" />
        <Setter Property="HorizontalOptions" Value="Center" />
        <Setter Property="Margin" Value="10" />
    </VisualState.Setters>
</VisualState>
</VisualStateGroup>
</VisualStateGroupList>
</Setter>
</Style>
</StackLayout.Resources>

<Button Text="Banana"
    Command="{Binding SelectedCommand}"
    CommandParameter="Banana.jpg" />

<Button Text="Face Palm"
    Command="{Binding SelectedCommand}"
    CommandParameter="FacePalm.jpg" />

<Button Text="Monkey"
    Command="{Binding SelectedCommand}"
    CommandParameter="monkey.png" />

<Button Text="Seated Monkey"
    Command="{Binding SelectedCommand}"
    CommandParameter="SeatedMonkey.jpg" />
</StackLayout>
</ScrollView>

<Image x:Name="image"
    VerticalOptions="FillAndExpand"
    HorizontalOptions="FillAndExpand" />
</StackLayout>
</ContentPage>

```

The inner `ScrollView` named `menuScroll` and the `StackLayout` named `menuStack` implement the menu of buttons. The orientation of these layouts is opposite of `mainStack`. The menu should be horizontal in portrait mode and vertical in landscape mode.

The fourth section of VSM markup is in an implicit style for the buttons themselves. This markup sets `VerticalOptions`, `HorizontalOptions`, and `Margin` properties specific to the portait and landscape orientations.

The code-behind file sets the `BindingContext` property of `menuStack` to implement `Button` commanding, and also attaches a handler to the `SizeChanged` event of the page:

```

public partial class VsmAdaptiveLayoutPage : ContentPage
{
    public VsmAdaptiveLayoutPage ()
    {
        InitializeComponent ();

        SizeChanged += (sender, args) =>
        {
            string visualState = Width > Height ? "Landscape" : "Portrait";
            VisualStateManager.GoToState(mainStack, visualState);
            VisualStateManager.GoToState(menuScroll, visualState);
            VisualStateManager.GoToState(menuStack, visualState);

            foreach (View child in menuStack.Children)
            {
                VisualStateManager.GoToState(child, visualState);
            }
        };
    }

    SelectedCommand = new Command<string>((filename) =>
    {
        image.Source = ImageSource.FromResource("VsmDemos.Images." + filename);
    });

    menuStack.BindingContext = this;
}

public ICommand SelectedCommand { private set; get; }
}

```

The `SizeChanged` handler calls `VisualStateManager.GoToState` for the two `StackLayout` and `ScrollView` elements, and then loops through the children of `menuStack` to call `VisualStateManager.GoToState` for the `Button` elements.

It may seem as if the code-behind file can handle orientation changes more directly by setting properties of elements in the XAML file, but the Visual State Manager is definitely a more structured approach. All the visuals are kept in the XAML file, where they become easier to examine, maintain, and modify.

## Visual State Manager with Xamarin.University

[Xamarin.Forms 3.0 Visual State Manager, by Xamarin University](#)

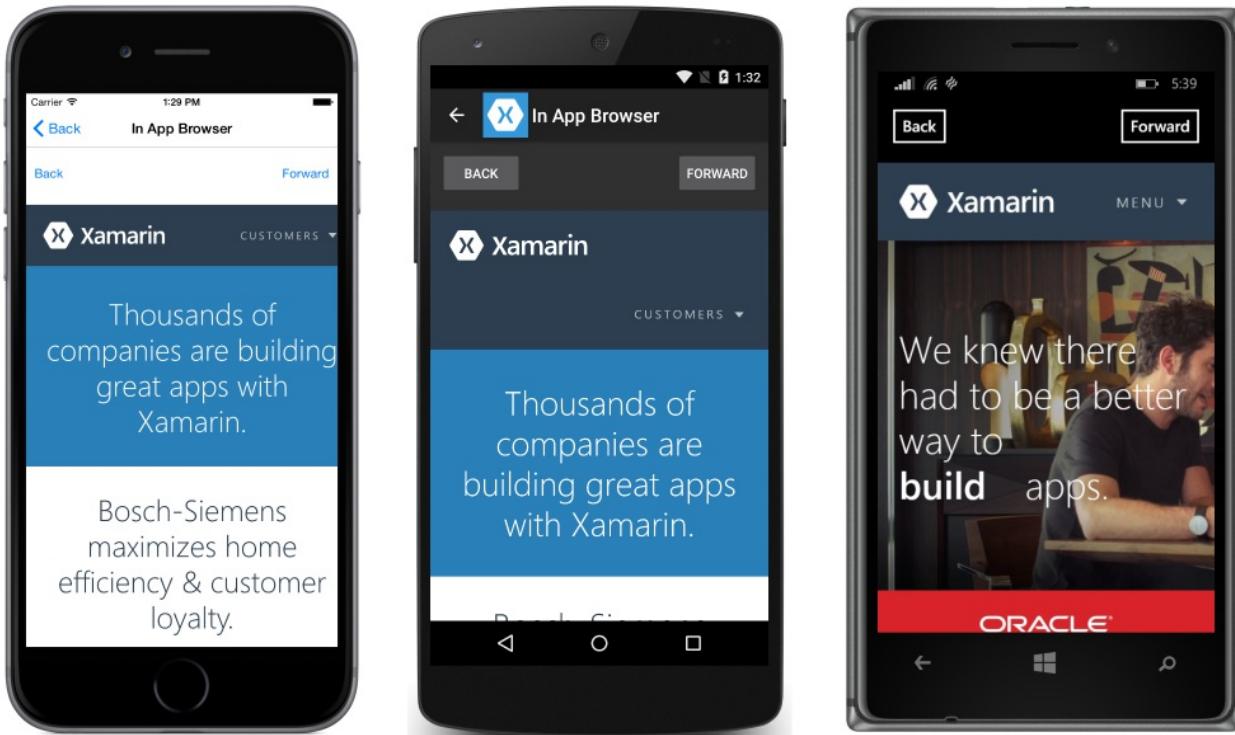
### Related links

- [VsmDemos](#)

# Xamarin.Forms WebView

11/20/2018 • 9 minutes to read • [Edit Online](#)

`WebView` is a view for displaying web and HTML content in your app. Unlike `openUri`, which takes the user to the web browser on the device, `WebView` displays the HTML content inside your app.



## Content

`WebView` supports the following types of content:

- HTML & CSS websites – `WebView` has full support for websites written using HTML & CSS, including JavaScript support.
- Documents – Because `WebView` is implemented using native components on each platform, `WebView` is capable of showing documents that are viewable on each platform. That means that PDF files work on iOS and Android.
- HTML strings – `WebView` can show HTML strings from memory.
- Local Files – `WebView` can present any of the content types above embedded in the app.

### NOTE

`WebView` on Windows does not support Silverlight, Flash or any ActiveX controls, even if they are supported by Internet Explorer on that platform.

## Websites

To display a website from the internet, set the `WebView`'s `Source` property to a string URL:

```
var browser = new WebView
{
    Source = "http://xamarin.com"
};
```

#### NOTE

URLs must be fully formed with the protocol specified (i.e. it must have "http://" or "https://" prepended to it).

#### iOS and ATS

Since version 9, iOS will only allow your application to communicate with servers that implement best-practice security by default. Values must be set in `Info.plist` to enable communication with insecure servers.

#### NOTE

If your application requires a connection to an insecure website, you should always enter the domain as an exception using `NSEExceptionDomains` instead of turning ATS off completely using `NSAllowsArbitraryLoads`. `NSAllowsArbitraryLoads` should only be used in extreme emergency situations.

The following demonstrates how to enable a specific domain (in this case xamarin.com) to bypass ATS requirements:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSEExceptionDomains</key>
    <dict>
        <key>xamarin.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
            <true/>
            <key>NSTemporaryExceptionMinimumTLSVersion</key>
            <string>TLSv1.1</string>
        </dict>
    </dict>
</dict>
...
</key>
```

It is best practice to only enable some domains to bypass ATS, allowing you to use trusted sites while benefitting from the additional security on untrusted domains. The following demonstrates the less secure method of disabling ATS for the app:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads </key>
    <true/>
</dict>
...
</key>
```

See [App Transport Security](#) for more information about this new feature in iOS 9.

#### HTML Strings

If you want to present a string of HTML defined dynamically in code, you'll need to create an instance of

```
HtmlWebViewSource :
```

```
var browser = new WebView();
var htmlSource = new HtmlWebViewSource();
htmlSource.Html = @"<html><body>
<h1>Xamarin.Forms</h1>
<p>Welcome to WebView.</p>
</body></html>";
browser.Source = htmlSource;
```



In the above code, `@` is used to mark the HTML as a string literal, meaning all the usual escape characters are ignored.

### Local HTML Content

WebView can display content from HTML, CSS and Javascript embedded within the app. For example:

```
<html>
<head>
    <title>Xamarin Forms</title>
</head>
<body>
    <h1>Xamarin.Forms</h1>
    <p>This is an iOS web page.</p>
    
</body>
</html>
```

CSS:

```
html, body {  
    margin: 0;  
    padding: 10;  
}  
body, p, h1 {  
    font-family: Chalkduster;  
}
```

Note that the fonts specified in the above CSS will need to be customized for each platform, as not every platform has the same fonts.

To display local content using a `WebView`, you'll need to open the HTML file like any other, then load the contents as a string into the `Html` property of an `HtmlWebViewSource`. For more information on opening files, see [Working with Files](#).

The following screenshots show the result of displaying local content on each platform:



Although the first page has been loaded, the `WebView` has no knowledge of where the HTML came from. That is a problem when dealing with pages that reference local resources. Examples of when that might happen include when local pages link to each other, a page makes use of a separate JavaScript file, or a page links to a CSS stylesheet.

To solve this, you need to tell the `WebView` where to find files on the filesystem. Do that by setting the `BaseUrl` property on the `HtmlWebViewSource` used by the `WebView`.

Because the filesystem on each of the operating systems is different, you need to determine that URL on each platform. Xamarin.Forms exposes the `DependencyService` for resolving dependencies at runtime on each platform.

To use the `DependencyService`, first define an interface that can be implemented on each platform:

```
public interface IBaseUrl { string Get(); }
```

Note that until the interface is implemented on each platform, the app will not run. In the common project, make sure that you remember to set the `BaseUrl` using the `DependencyService`:

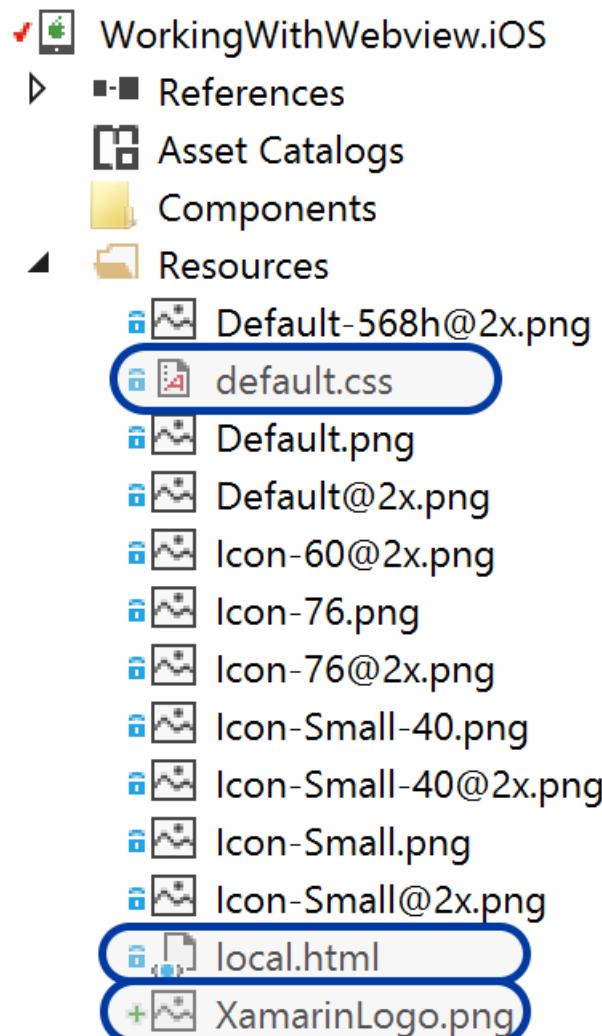
```
var source = new HtmlWebViewSource();
source.BaseUrl = DependencyService.Get<IBaseUrl>().Get();
```

Implementations of the interface for each platform must then be provided.

#### iOS

On iOS, the web content should be located in the project's root directory or **Resources** directory with build action *BundleResource*, as demonstrated below:

- [Visual Studio](#)
- [Visual Studio for Mac](#)



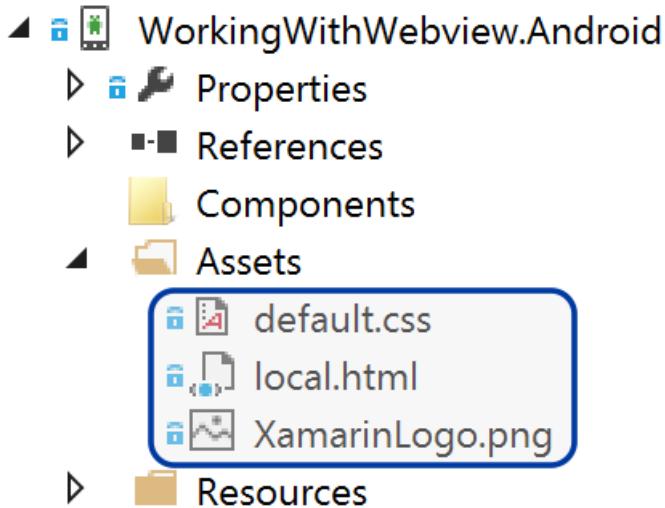
The `BaseUrl` should be set to the path of the main bundle:

```
[assembly: Dependency (typeof (BaseUrl_iOS))]
namespace WorkingWithWebview.iOS
{
    public class BaseUrl_iOS : IBaseUrl
    {
        public string Get()
        {
            return NSBundle.MainBundle.BundlePath;
        }
    }
}
```

## Android

On Android, place HTML, CSS, and images in the Assets folder with build action *AndroidAsset* as demonstrated below:

- [Visual Studio](#)
- [Visual Studio for Mac](#)



On Android, the `BaseUrl` should be set to `"file:///android_asset/"`:

```
[assembly: Dependency (typeof(BaseUrl_Android))]
namespace WorkingWithWebview.Android
{
    public class BaseUrl_Android : IBaseUrl
    {
        public string Get()
        {
            return "file:///android_asset/";
        }
    }
}
```

On Android, files in the **Assets** folder can also be accessed through the current Android context, which is exposed by the `MainActivity.Instance` property:

```
var assetManager = MainActivity.Instance.Assets;
using (var streamReader = new StreamReader (assetManager.Open ("local.html")))
{
    var html = streamReader.ReadToEnd ();
}
```

On Universal Windows Platform (UWP) projects, place HTML, CSS and images in the project root with the build action set to *Content*.

The `BaseUrl` should be set to `"ms-appx-web:///"`:

```
[assembly: Dependency(typeof(BaseUrl))]
namespace WorkingWithWebView.UWP
{
    public class BaseUrl : IBaseUrl
    {
        public string Get()
        {
            return "ms-appx-web:///";
        }
    }
}
```

## Navigation

WebView supports navigation through several methods and properties that it makes available:

- **GoForward()** – if `CanGoForward` is true, calling `GoForward` navigates forward to the next visited page.
- **GoBack()** – if `CanGoBack` is true, calling `GoBack` will navigate to the last visited page.
- **CanGoBack** – `true` if there are pages to navigate back to, `false` if the browser is at the starting URL.
- **CanGoForward** – `true` if the user has navigated backwards and can move forward to a page that was already visited.

Within pages, `WebView` does not support multi-touch gestures. It is important to make sure that content is mobile-optimized and appears without the need for zooming.

It is common for applications to show a link within a `WebView`, rather than the device's browser. In those situations, it is useful to allow normal navigation, but when the user hits back while they are on the starting link, the app should return to the normal app view.

Use the built-in navigation methods and properties to enable this scenario.

Start by creating the page for the browser view:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WebViewSample.InAppBrowserXaml"
             Title="Browser">
    <StackLayout Margin="20">
        <StackLayout Orientation="Horizontal">
            <Button Text="Back" HorizontalOptions="StartAndExpand" Clicked="OnBackButtonClicked" />
            <Button Text="Forward" HorizontalOptions="EndAndExpand" Clicked="OnForwardButtonClicked" />
        </StackLayout>
        <!-- WebView needs to be given height and width request within layouts to render. -->
        <WebView x:Name="webView" WidthRequest="1000" HeightRequest="1000" />
    </StackLayout>
</ContentPage>
```

In the code-behind:

```

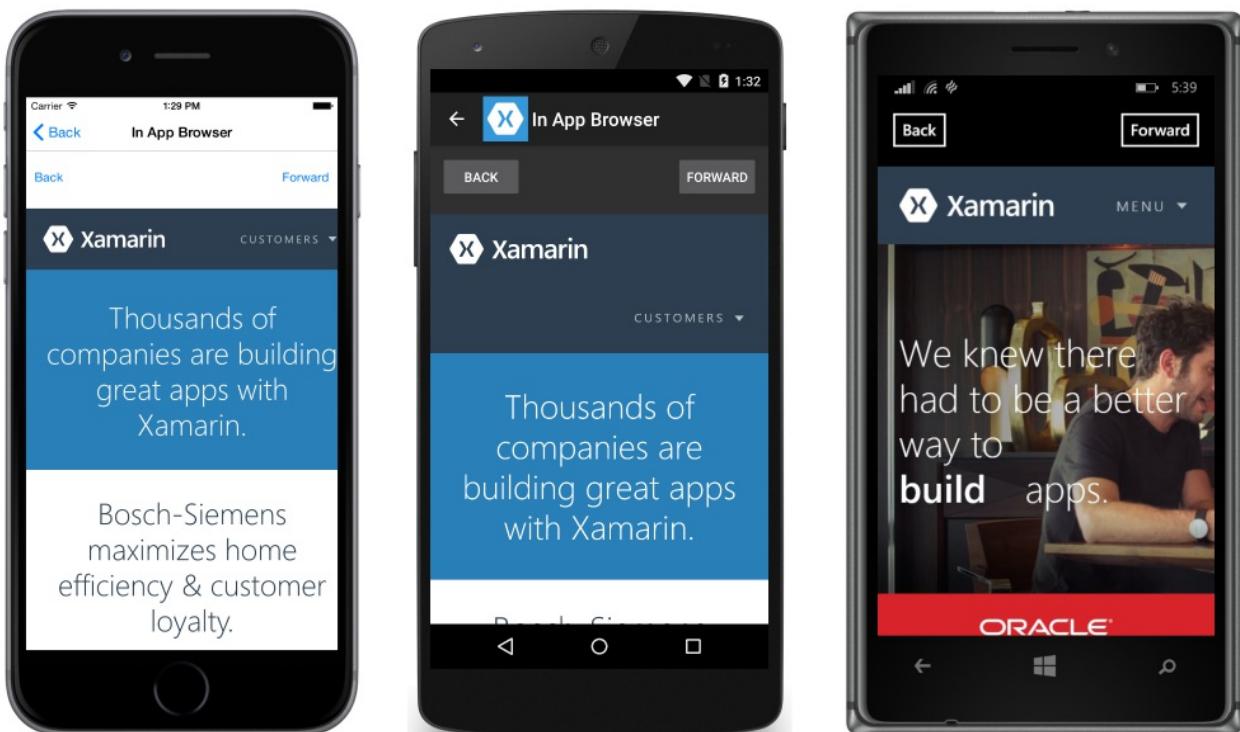
public partial class InAppBrowserXaml : ContentPage
{
    public InAppBrowserXaml(string URL)
    {
        InitializeComponent();
        webView.Source = URL;
    }

    async void OnBackButtonClicked(object sender, EventArgs e)
    {
        if (webView.CanGoBack)
        {
            webView.GoBack();
        }
        else
        {
            await Navigation.PopAsync();
        }
    }

    void OnForwardButtonClicked(object sender, EventArgs e)
    {
        if (webView.CanGoForward)
        {
            webView.GoForward();
        }
    }
}

```

That's it!



## Events

WebView raises the following events to help you respond to changes in state:

- **Navigating** – event raised when the WebView begins loading a new page.
- **Navigated** – event raised when the page is loaded and navigation has stopped.
- **ReloadRequested** – event raised when a request is made to reload the current content.

If you anticipate using webpages that take a long time to load, consider using the `Navigating` and `Navigated` events to implement a status indicator. For example the XAML looks like this:

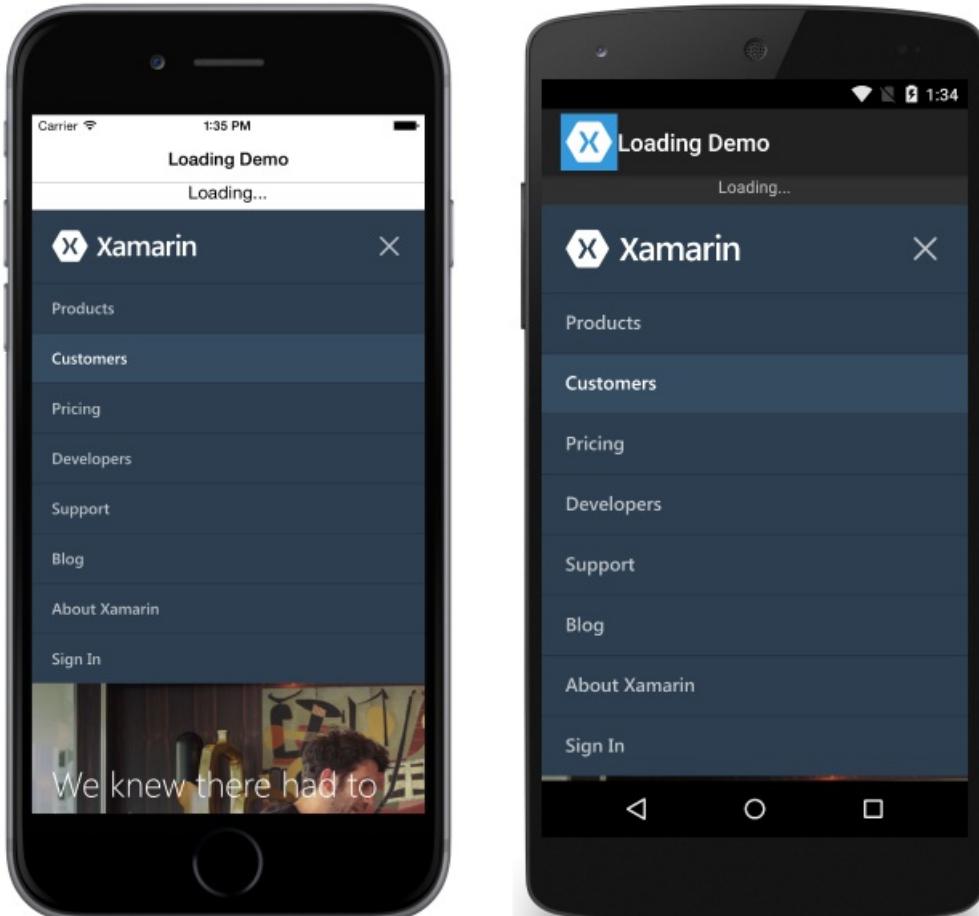
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WebViewSample.LoadingLabelXaml"
    Title="Loading Demo">
<StackLayout>
    <!--Loading label should not render by default.-->
    <Label x:Name="labelLoading" Text="Loading..." IsVisible="false" />
    <WebView HeightRequest="1000" WidthRequest="1000" Source="http://www.xamarin.com"
    Navigated="webViewNavigated" Navigating="webViewNavigating" />
</StackLayout>
</ContentPage>
```

The two event handlers:

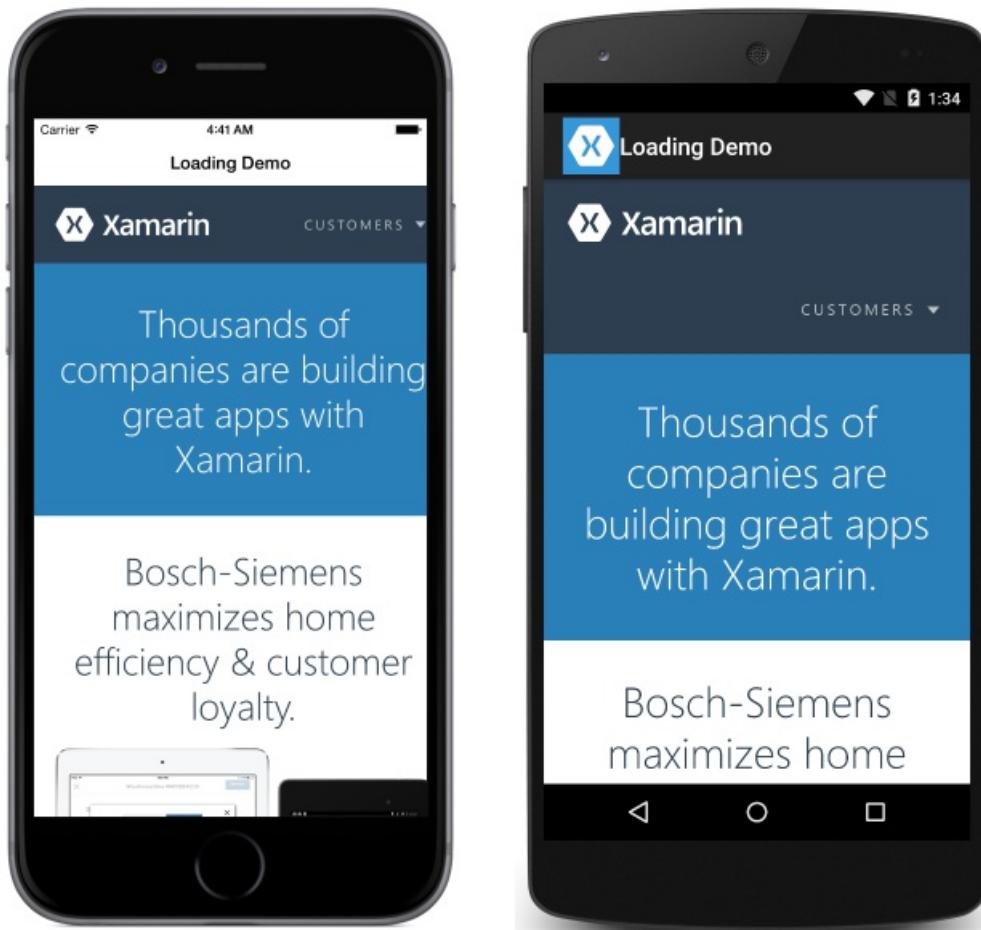
```
void webViewNavigating(object sender, WebNavigatingEventArgs e)
{
    labelLoading.IsVisible = true;
}

void webViewNavigated(object sender, WebNavigatedEventArgs e)
{
    labelLoading.IsVisible = false;
}
```

This results in the following output (loading):



Finished Loading:



## Reloading content

`WebView` has a `Reload` method that can be used to reload the current content:

```
var webView = new WebView();
...
webView.Reload();
```

When the `Reload` method is invoked the `ReloadRequested` event is fired, indicating that a request has been made to reload the current content.

## Performance

The popular web browsers now adopt technologies like hardware accelerated rendering and JavaScript compilation. On iOS, by default, the Xamarin.Forms `WebView` is implemented by the `UIWebView` class, and many of these technologies are unavailable in this implementation. However, an application can opt-in to using the iOS `WKWebView` class to implement the Xamarin.Forms `WebView`, which supports faster browsing. This can be achieved by adding the following code to the **AssemblyInfo.cs** file in the iOS platform project for the application:

```
// Opt-in to using WKWebView instead of UIWebView.
[assembly: ExportRenderer(typeof(WebView), typeof(Xamarin.Forms.Platform.iOS.WkWebViewRenderer))]
```

`WebView` on Android by default is about as fast as the built-in browser.

The [UWP WebView](#) uses the Microsoft Edge rendering engine. Desktop and tablet devices should see the same performance as using the Edge browser itself.

## Permissions

In order for `WebView` to work, you must make sure that permissions are set for each platform. Note that on some platforms, `WebView` will work in debug mode, but not when built for release. That is because some permissions, like those for internet access on Android, are set by default by Visual Studio for Mac when in debug mode.

- **UWP** – requires the Internet (Client & Server) capability when displaying network content.
- **Android** – requires `INTERNET` only when displaying content from the network. Local content requires no special permissions.
- **iOS** – requires no special permissions.

## Layout

Unlike most other Xamarin.Forms views, `WebView` requires that `HeightRequest` and `WidthRequest` are specified when contained in `StackLayout` or `RelativeLayout`. If you fail to specify those properties, the `WebView` will not render.

The following examples demonstrate layouts that result in working, rendering `WebView`s:

`StackLayout` with `WidthRequest` & `HeightRequest`:

```
<StackLayout>
    <Label Text="test" />
    <WebView Source="http://www.xamarin.com/"
        HeightRequest="1000"
        WidthRequest="1000" />
</StackLayout>
```

`RelativeLayout` with `WidthRequest` & `HeightRequest`:

```
<RelativeLayout>
    <Label Text="test"
        RelativeLayout.XConstraint= "{ConstraintExpression
            Type=Constant, Constant=10}"
        RelativeLayout.YConstraint= "{ConstraintExpression
            Type=Constant, Constant=20}" />
    <WebView Source="http://www.xamarin.com/"
        RelativeLayout.XConstraint="{ConstraintExpression Type=Constant,
            Constant=10}"
        RelativeLayout.YConstraint="{ConstraintExpression Type=Constant,
            Constant=50}"
        WidthRequest="1000" HeightRequest="1000" />
</RelativeLayout>
```

`AbsoluteLayout` without `WidthRequest` & `HeightRequest`:

```
<AbsoluteLayout>
    <Label Text="test" AbsoluteLayout.LayoutBounds="0,0,100,100" />
    <WebView Source="http://www.xamarin.com/"
        AbsoluteLayout.LayoutBounds="0,150,500,500" />
</AbsoluteLayout>
```

Grid without `WidthRequest` & `HeightRequest`. Grid is one of the few layouts that does not require specifying requested heights and widths.:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Label Text="test" Grid.Row="0" />
    <WebView Source="http://www.xamarin.com/" Grid.Row="1" />
</Grid>
```

## Invoking JavaScript

The `WebView` includes the ability to invoke a JavaScript function from C#, and return any result to the calling C# code. This is accomplished with the `WebView.EvaluateJavaScriptAsync` method, which is shown in the following example from the [WebView](#) sample:

```
var numberEntry = new Entry { Text = "5" };
var resultLabel = new Label();
var webView = new WebView();
...

int number = int.Parse(numberEntry.Text);
string result = await webView.EvaluateJavaScriptAsync($"factorial({number})");
resultLabel.Text = $"Factorial of {number} is {result}.";
```

The `WebView.EvaluateJavaScriptAsync` method evaluates the JavaScript that's specified as the argument, and returns any result as a `string`. In this example, the `factorial` JavaScript function is invoked, which returns the factorial of `number` as a result. This JavaScript function is defined in the local HTML file that the `WebView` loads, and is shown in the following example:

```
<html>
<body>
<script type="text/javascript">
function factorial(num) {
    if (num === 0 || num === 1)
        return 1;
    for (var i = num - 1; i >= 1; i--) {
        num *= i;
    }
    return num;
}
</script>
</body>
</html>
```

## Related Links

- [Working with WebView \(sample\)](#)
- [WebView \(sample\)](#)

# Xamarin.Forms Platform Features

10/9/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms is extensible and lets you incorporate platform-specific features using [effects](#), [custom renderers](#), the [DependencyService](#), the [MessagingCenter](#), and more.

## Android

This guide describes how to implement Material Design by updating existing Xamarin.Forms Android apps.

## Application Indexing and Deep Linking

Application indexing allows applications that would otherwise be forgotten after a few uses to stay relevant by appearing in search results. Deep linking allows applications to respond to a search result that contains application data, typically by navigating to a page referenced from a deep link.

## Device Class

How to use the `Device` class to create platform-specific behavior in shared code and the user interface (including using XAML). Also covers `BeginInvokeOnMainThread` which is essential when modifying UI controls from background threads.

## iOS

Some iOS styling can be performed via `Info.plist` and the `UIAppearance` API. This guide includes examples of how to include iOS 9 features into the iOS app of a Xamarin.Forms solution, including Core Spotlight search.

## GTK

Xamarin.Forms now has preview support for GTK# apps.

## Mac

Xamarin.Forms now has preview support for macOS apps.

## Native Forms

Native Forms allow Xamarin.Forms `ContentPage`-derived pages to be consumed by native Xamarin.iOS, Xamarin.Android, and Universal Windows Platform (UWP) projects.

## Native Views

Native views from iOS, Android, and the Universal Windows Platform can be directly referenced from Xamarin.Forms. Properties and event handlers can be set on native views, and they can interact with Xamarin.Forms views.

## Platform-Specifics

Platform-specifics allow you to consume functionality that's only available on a specific platform, without requiring custom renderers or effects.

## Plugins

There are a wide variety of open-source plug-ins available on Github, Nuget, and the Xamarin Component Store to help extend Xamarin.Forms apps.

## Tizen

Tizen .NET enables you to build .NET applications with Xamarin.Forms and the Tizen .NET Framework.

## Windows

Xamarin.Forms has support for the Universal Windows Platform (UWP) on Windows 10. This article describes how to add a UWP project to an existing Xamarin.Forms solution.

## WPF

Xamarin.Forms now has preview support for Windows Presentation Foundation (WPF) apps.

# Android Platform Features

6/14/2018 • 2 minutes to read • [Edit Online](#)

## Platform Support

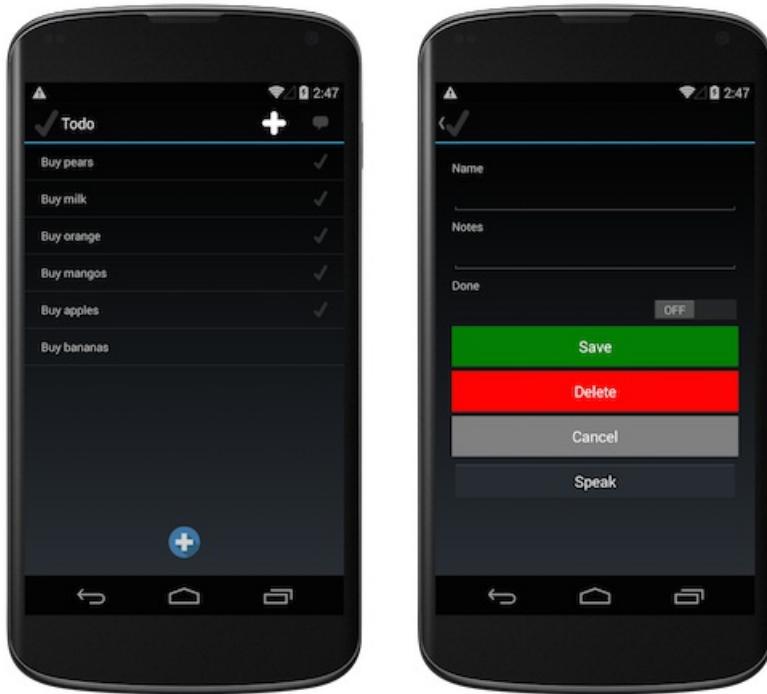
Originally, the default Xamarin.Forms Android project used an older style of control rendering that was common prior to Android 5.0. Applications built using the template have `FormsApplicationActivity` as the base class of their main activity.

## Material Design via AppCompat

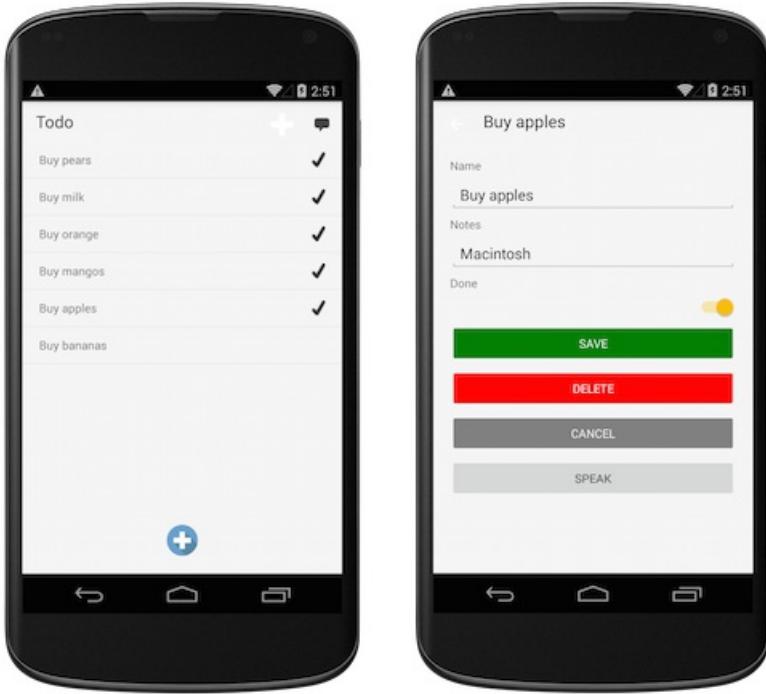
Xamarin.Forms Android projects now use `FormsAppCompatActivity` as the base class of their main activity. This class uses **AppCompat** features provided by Android to implement Material Design themes.

To add Material Design themes to your Xamarin.Forms Android project, follow the [installation instructions for AppCompat support](#)

Here is the **Todo** sample with the default `FormsApplicationActivity` :



And this is the same code after upgrading the project to use `FormsAppCompatActivity` (and adding the additional theme information):



#### NOTE

When using `FormsAppCompatActivity`, the base classes for some Android custom renderers will be different.

## Related Links

- [Add Material Design Support](#)

# Adding AppCompat and Material Design

6/8/2018 • 2 minutes to read • [Edit Online](#)

Follow these steps to convert existing Xamarin.Forms Android apps to use AppCompat and Material Design

## Overview

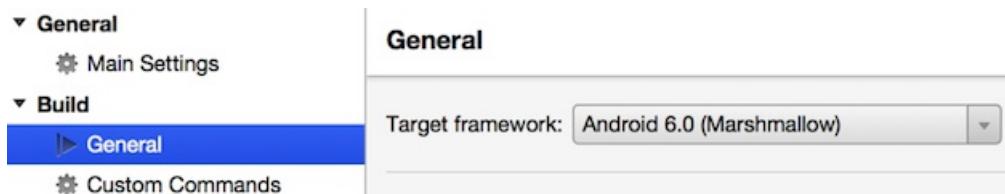
These instructions explain how to update your existing Xamarin.Forms Android applications to use the AppCompat library and enable Material Design in the Android version of your Xamarin.Forms apps.

### 1. Update Xamarin.Forms

Ensure the solution is using Xamarin.Forms 2.0 or newer. Update the Xamarin.Forms Nuget package to 2.0 if required.

### 2. Check Android version

Ensure the Android project's target framework is Android 6.0 (Marshmallow). Check the **Android project > Options > Build > General** settings to ensure the current framework is selected:



### 3. Add new themes to support Material Design

Create the following three files in your Android project and paste in the contents below. Google provides a [style guide](#) and a [color palette generator](#) to help you choose an alternate color scheme to the one specified.

#### Resources/values/colors.xml

```
<resources>
    <color name="primary">#2196F3</color>
    <color name="primaryDark">#1976D2</color>
    <color name="accent">#FFC107</color>
    <color name="window_background">#F5F5F5</color>
</resources>
```

#### Resources/values/style.xml

```
<resources>
    <style name="MyTheme" parent="MyTheme.Base">
    </style>
    <style name="MyTheme.Base" parent="Theme.AppCompat.Light.NoActionBar">
        <item name="colorPrimary">@color/primary</item>
        <item name="colorPrimaryDark">@color/primaryDark</item>
        <item name="colorAccent">@color/accent</item>
        <item name="android:windowBackground">@color/window_background</item>
        <item name="windowActionBarOverlay">true</item>
    </style>
</resources>
```

An additional style must be included in the **values-v21** folder to apply specific properties when running on Android Lollipop and newer.

## Resources/values-v21/style.xml

```
<resources>
    <style name="MyTheme" parent="MyTheme.Base">
        <!--If you are using MasterDetailPage you will want to set these, else you can leave them out-->
        <!--<item name="android:windowDrawsSystemBarBackgrounds">true</item>
        <item name="android:statusBarColor">@android:color/transparent</item>-->
    </style>
</resources>
```

## 4. Update AndroidManifest.xml

To ensure this new theme information is used, set theme in the **AndroidManifest** file by adding

```
    android:theme="@style/MyTheme" (leave the rest of the XML as it was).
```

## Properties/AndroidManifest.xml

```
...
<application android:label="AppName" android:icon="@drawable/icon"
    android:theme="@style/MyTheme">
    ...

```

## 5. Provide toolbar and tab layouts

Create **Tabbar.axml** and **Toolbar.axml** files in the **Resources/layout** directory and paste in their contents from below:

### Resources/layout/Tabbar.axml

```
<android.support.design.widget.TabLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/sliding_tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:tabIndicatorColor="@android:color/white"
    app:tabGravity="fill"
    app:tabMode="fixed" />
```

A few properties for the tabs have been set including the tab's gravity to `fill` and mode to `fixed`. If you have a lot of tabs you may want to switch this to scrollable - read through the [Android TabLayout documentation](#) to learn more.

### Resources/layout/Toolbar.axml

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:minHeight="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
    app:layout_scrollFlags="scroll|enterAlways" />
```

In these files we're creating specific theme for the toolbar that may vary for your application. Refer to the [Hello](#)

[Toolbar](#) blog post to learn more.

## 6. Update the `MainActivity`

In existing Xamarin.Forms apps the **MainActivity.cs** class will inherit from `FormsApplicationActivity`. This must be replaced with `FormsAppCompatActivity` to enable the new functionality.

### MainActivity.cs

```
public class MainActivity : AppCompatActivity // was FormsApplicationActivity
```

Finally, "wire up" the new layouts from step 5 in the `OnCreate` method, as shown here:

```
protected override void OnCreate(Bundle bundle)
{
    // set the layout resources first
    AppCompatActivity.ToolbarResource = Resource.Layout.Toolbar;
    AppCompatActivity.TabLayoutResource = Resource.Layout.Tabbar;

    // then call base.OnCreate and the Xamarin.Forms methods
    base.OnCreate(bundle);
    Forms.Init(this, bundle);
    LoadApplication(new App());
}
```

# Application Indexing and Deep Linking

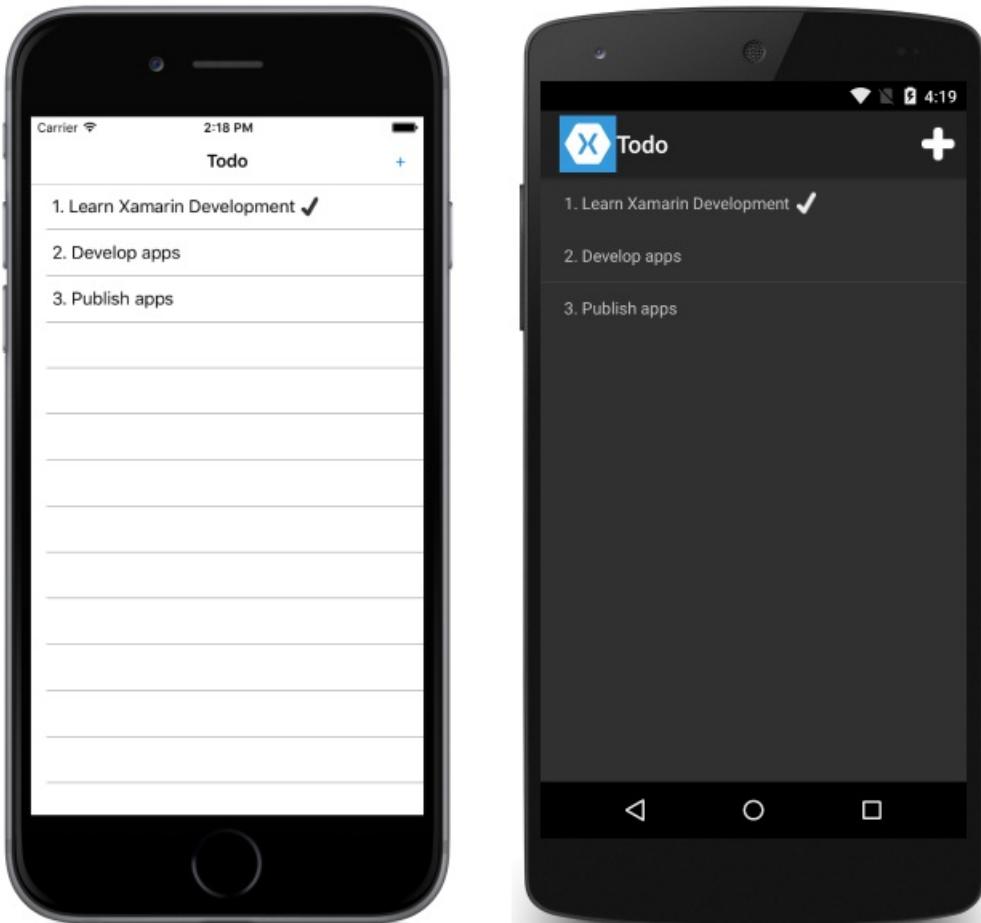
7/12/2018 • 8 minutes to read • [Edit Online](#)

*Application indexing allows applications that would otherwise be forgotten after a few uses to stay relevant by appearing in search results. Deep linking allows applications to respond to a search result that contains application data, typically by navigating to a page referenced from a deep link. This article demonstrates how to use application indexing and deep linking to make Xamarin.Forms application content searchable on iOS and Android devices.*

## Deep Linking with Xamarin.Forms and Azure, by [Xamarin University](#)

Xamarin.Forms application indexing and deep linking provide an API for publishing metadata for application indexing as users navigate through applications. Indexed content can then be searched for in Spotlight Search, in Google Search, or in a web search. Tapping on a search result that contains a deep link will fire an event that can be handled by an application, and is typically used to navigate to the page referenced from the deep link.

The sample application demonstrates a Todo list application where the data is stored in a local SQLite database, as shown in the following screenshots:



Each `TodoItem` instance created by the user is indexed. Platform-specific search can then be used to locate indexed data from the application. When the user taps on a search result item for the application, the application is launched, the `TodoItemPage` is navigated to, and the `TodoItem` referenced from the deep link is displayed.

For more information about using an SQLite database, see [Working with a Local Database](#).

## NOTE

Xamarin.Forms application indexing and deep linking functionality is only available on the iOS and Android platforms, and requires iOS 9 and API 23 respectively.

# Setup

The following sections provide any additional setup instructions for using this feature on the iOS and Android platforms.

## iOS

On the iOS platform, there's no additional setup required to use this functionality.

## Android

On the Android platform, there are a number of prerequisites that must be met to use application indexing and deep linking functionality:

1. A version of your application must be live on Google Play.
2. A companion website must be registered against the application in Google's Developer Console. Once the application is associated with a website, URLs can be indexed that work for both the website and the application, which can then be served in Search results. For more information, see [App Indexing on Google Search](#) on Google's website.
3. Your application must support HTTP URL intents on the `MainActivity` class, which tell application indexing what types of URL data schemes the application can respond to. For more information, see [Configuring the Intent Filter](#).

Once these prerequisites are met, the following additional setup is required to use Xamarin.Forms application indexing and deep linking on the Android platform:

1. Install the [Xamarin.Forms.AppLinks](#) NuGet package into the Android application project.
2. In the `MainActivity.cs` file, import the `Xamarin.Forms.Platform.Android.AppLinks` namespace.
3. In the `MainActivity.OnCreate` override, add the following line of code underneath `Forms.Init(this, bundle)`:

```
AndroidAppLinks.Init (this);
```

For more information, see [Deep Link Content with Xamarin.Forms URL Navigation](#) on the Xamarin blog.

# Indexing a Page

The process for indexing a page and exposing it to Google and Spotlight search is as follows:

1. Create an `AppLinkEntry` that contains the metadata required to index the page, along with a deep link to return to the page when the user selects the indexed content in search results.
2. Register the `AppLinkEntry` instance to index it for searching.

The following code example demonstrates how to create an `AppLinkEntry` instance:

```

AppLinkEntry GetAppLink (TodoItem item)
{
    var pageType = GetType ().ToString ();
    var pageLink = new AppLinkEntry {
        Title = item.Name,
        Description = item.Notes,
        AppLinkUri = new Uri (string.Format ("http://{0}/{1}?id={2}",
            App.AppName, pageType, WebUtility.UrlEncode (item.ID)), UriKind.RelativeOrAbsolute),
        IsLinkActive = true,
        Thumbnail = ImageSource.FromFile ("monkey.png")
    };

    return pageLink;
}

```

The `AppLinkEntry` instance contains a number of properties whose values are required to index the page and create a deep link. The `Title`, `Description`, and `Thumbnail` properties are used to identify the indexed content when it appears in search results. The `IsLinkActive` property is set to `true` to indicate that the indexed content is currently being viewed. The `AppLinkUri` property is a `Uri` that contains the information required to return to the current page and display the current `TodoItem`. The following example shows an example `Uri` for the sample application:

```
http://deeplinking/DeepLinking.TodoItemPage?id=ec38ebd1-811e-4809-8a55-0d028fce7819
```

This `Uri` contains all the information required to launch the `deeplinking` app, navigate to the `DeepLinking.TodoItemPage`, and display the `TodoItem` that has an `ID` of `ec38ebd1-811e-4809-8a55-0d028fce7819`.

## Registering Content for Indexing

Once an `AppLinkEntry` instance has been created, it must be registered for indexing to appear in search results. This is accomplished with the `RegisterLink` method, as demonstrated in the following code example:

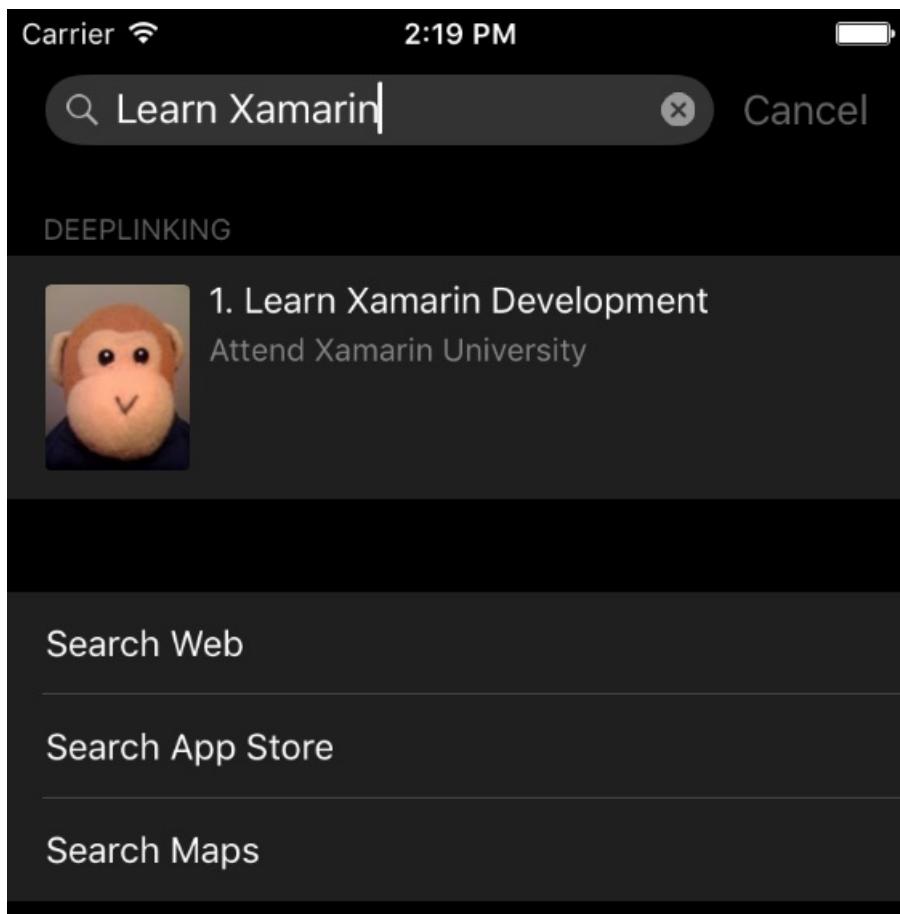
```
Application.Current.AppLinks.RegisterLink (appLink);
```

This adds the `AppLinkEntry` instance to the application's `AppLinks` collection.

### NOTE

The `RegisterLink` method can also be used to update the content that's been indexed for a page.

Once an `AppLinkEntry` instance has been registered for indexing, it can appear in search results. The following screenshot shows indexed content appearing in search results on the iOS platform:



## De-registering Indexed Content

The `DeregisterLink` method is used to remove indexed content from search results, as demonstrated in the following code example:

```
Application.Current.AppLinks.DeregisterLink (appLink);
```

This removes the `AppLinkEntry` instance from the application's `AppLinks` collection.

### NOTE

On Android it's not possible to remove indexed content from search results.

## Responding to a Deep Link

When indexed content appears in search results and is selected by a user, the `App` class for the application will receive a request to handle the `Uri` contained in the indexed content. This request can be processed in the `OnAppLinkRequestReceived` override, as demonstrated in the following code example:

```

public class App : Application
{
    ...

    protected override async void OnAppLinkRequestReceived (Uri uri)
    {
        string appDomain = "http://" + App.AppName.ToLowerInvariant () + "/";
        if (!uri.ToString ().ToLowerInvariant ().StartsWith (appDomain)) {
            return;
        }

        string pageUrl = uri.ToString ().Replace (appDomain, string.Empty).Trim ();
        var parts = pageUrl.Split ('?');
        string page = parts [0];
        string pageParameter = parts [1].Replace ("id=", string.Empty);

        var formsPage = Activator.CreateInstance (Type.GetType (page));
        var todoItemPage = formsPage as TodoItemPage;
        if (todoItemPage != null) {
            var todoItem = App.Database.Find (pageParameter);
            todoItemPage.BindingContext = todoItem;
            await MainPage.Navigation.PushAsync (formsPage as Page);
        }
    }

    base.OnAppLinkRequestReceived (uri);
}
}

```

The `OnAppLinkRequestReceived` method checks that the received `Uri` is intended for the application, before parsing the `Uri` into the page to be navigated to and the parameter to be passed to the page. An instance of the page to be navigated to is created, and the `TodoItem` represented by the page parameter is retrieved. The `BindingContext` of the page to be navigated to is then set to the `TodoItem`. This ensures that when the `TodoItemPage` is displayed by the `PushAsync` method, it will be showing the `TodoItem` whose `ID` is contained in the deep link.

## Making Content Available for Search Indexing

Each time the page represented by a deep link is displayed, the `AppLinkEntry.IsActive` property can be set to `true`. On iOS and Android this makes the `AppLinkEntry` instance available for search indexing, and on iOS only, it also makes the `AppLinkEntry` instance available for Handoff. For more information about Handoff, see [Introduction to Handoff](#).

The following code example demonstrates setting the `AppLinkEntry.IsActive` property to `true` in the `Page.OnAppearing` override:

```

protected override void OnAppearing ()
{
    appLink = GetAppLink (BindingContext as TodoItem);
    if (appLink != null) {
        appLink.IsActive = true;
    }
}

```

Similarly, when the page represented by a deep link is navigated away from, the `AppLinkEntry.IsActive` property can be set to `false`. On iOS and Android, this stops the `AppLinkEntry` instance being advertised for search indexing, and on iOS only, it also stops advertising the `AppLinkEntry` instance for Handoff. This can be accomplished in the `Page.OnDisappearing` override, as demonstrated in the following code example:

```
protected override void OnDisappearing ()
{
    if (appLink != null) {
        appLink.IsLinkActive = false;
    }
}
```

## Providing Data to Handoff

On iOS, application-specific data can be stored when indexing the page. This is achieved by adding data to the `KeyValues` collection, which is a `Dictionary<string, string>` for storing key-value pairs that are used in Handoff. Handoff is a way for the user to start an activity on one of their devices and continue that activity on another of their devices (as identified by the user's iCloud account). The following code shows an example of storing application-specific key-value pairs:

```
var pageLink = new AppLinkEntry {
    ...
};

pageLink.KeyValues.Add("appName", App.AppName);
pageLink.KeyValues.Add("companyName", "Xamarin");
```

Values stored in the `KeyValues` collection will be stored in the metadata for the indexed page, and will be restored when the user taps on a search result that contains a deep link (or when Handoff is used to view the content on another signed-in device).

In addition, values for the following keys can be specified:

- `contentType` – a `string` that specifies the uniform type identifier of the indexed content. The recommended convention to use for this value is the type name of the page containing the indexed content.
- `associatedWebPage` – a `string` that represents the web page to visit if the indexed content can also be viewed on the web, or if the application supports Safari's deep links.
- `shouldAddToPublicIndex` – a `string` of either `true` or `false` that controls whether or not to add the indexed content to Apple's public cloud index, which can then be presented to users who haven't installed the application on their iOS device. However, just because content has been set for public indexing, it doesn't mean that it will be automatically added to Apple's public cloud index. For more information, see [Public Search Indexing](#). Note that this key should be set to `false` when adding personal data to the `KeyValues` collection.

### NOTE

The `KeyValues` collection isn't used on the Android platform.

For more information about Handoff, see [Introduction to Handoff](#).

## Summary

This article demonstrated how to use application indexing and deep linking to make Xamarin.Forms application content searchable on iOS and Android devices. Application indexing allows applications to stay relevant by appearing in search results that would otherwise be forgotten about after a few uses.

## Related Links

- [Deep Linking \(sample\)](#)
- [iOS Search APIs](#)

- [App-Linking in Android 6.0](#)
- [AppLinkEntry](#)
- [IAppLinkEntry](#)
- [IAppLinks](#)

# Xamarin.Forms Device Class

9/20/2018 • 4 minutes to read • [Edit Online](#)

The `Device` class contains a number of properties and methods to help developers customize layout and functionality on a per-platform basis.

In addition to methods and properties to target code at specific hardware types and sizes, the `Device` class includes the `BeginInvokeOnMainThread` method which should be used when interacting with UI controls from background threads.

## Providing Platform-Specific Values

Prior to Xamarin.Forms 2.3.4, the platform the application was running on could be obtained by examining the `Device.OS` property and comparing it to the `TargetPlatform.iOS`, `TargetPlatform.Android`, `TargetPlatform.WinPhone`, and `TargetPlatform.Windows` enumeration values. Similarly, one of the `Device.OnPlatform` overloads could be used to provide platform-specific values to a control.

However, since Xamarin.Forms 2.3.4 these APIs have been deprecated and replaced. The `Device` class now contains public string constants that identify platforms – `Device.iOS`, `Device.Android`, `Device.WinPhone` (deprecated), `Device.WinRT` (deprecated), `Device.UWP`, and `Device.macOS`. Similarly, the `Device.OnPlatform` overloads have been replaced with the `OnPlatform` and `On` APIs.

In C#, platform-specific values can be provided by creating a `switch` statement on the `Device.RuntimePlatform` property, and then providing `case` statements for the required platforms:

```
double top;
switch (Device.RuntimePlatform)
{
    case Device.iOS:
        top = 20;
        break;
    case Device.Android:
    case Device.UWP:
    default:
        top = 0;
        break;
}
layout.Margin = new Thickness(5, top, 5, 0);
```

The `OnPlatform` and `On` classes provide the same functionality in XAML:

```
<StackLayout>
<StackLayout.Margin>
<OnPlatform x:TypeArguments="Thickness">
    <On Platform="iOS" Value="0,20,0,0" />
    <On Platform="Android, UWP" Value="0,0,0,0" />
</OnPlatform>
</StackLayout.Margin>
...
</StackLayout>
```

The `OnPlatform` class is a generic class that must be instantiated with an `x:TypeArguments` attribute that matches the target type. In the `On` class, the `Platform` attribute can accept a single `string` value, or multiple comma-

delimited `string` values.

#### IMPORTANT

Providing an incorrect `Platform` attribute value in the `On` class will not result in an error. Instead, the code will execute without the platform-specific value being applied.

Alternatively, the `OnPlatform` markup extension can be used in XAML to customize UI appearance on a per-platform basis. For more information, see [OnPlatform Markup Extension](#).

## Device.Idiom

The `Device.Idiom` property can be used to alter layouts or functionality depending on the device the application is running on. The `TargetIdiom` enumeration contains the following values:

- **Phone** – iPhone, iPod touch, and Android devices narrower than 600 dips<sup>^</sup>
- **Tablet** – iPad, Windows devices, and Android devices wider than 600 dips<sup>^</sup>
- **Desktop** – only returned in [UWP apps](#) on Windows 10 desktop computers (returns `Phone` on mobile Windows devices, including in Continuum scenarios)
- **TV** – Tizen TV devices
- **Watch** – Tizen watch devices
- **Unsupported** – unused

<sup>^</sup> *dips is not necessarily the physical pixel count*

The `Idiom` property is especially useful for building layouts that take advantage of larger screens, like this:

```
if (Device.Idiom == TargetIdiom.Phone) {
    // layout views vertically
} else {
    // layout views horizontally for a larger display (tablet or desktop)
}
```

The `OnIdiom` class provides the same functionality in XAML:

```
<StackLayout>
    <StackLayout.Margin>
        <OnIdiom x:TypeArguments="Thickness">
            <OnIdiom.Phone>0,20,0,0</OnIdiom.Phone>
            <OnIdiom.Tablet>0,40,0,0</OnIdiom.Tablet>
            <OnIdiom.Desktop>0,60,0,0</OnIdiom.Desktop>
        </OnIdiom>
    </StackLayout.Margin>
    ...
</StackLayout>
```

The `OnIdiom` class is a generic class that must be instantiated with an `x:TypeArguments` attribute that matches the target type.

Alternatively, the `OnIdiom` markup extension can be used in XAML to customize UI appearance based on the idiom of the device the application is running on. For more information, see [OnIdiom Markup Extension](#).

## Device.FlowDirection

The `Device.FlowDirection` value retrieves a `FlowDirection` enumeration value that represents the current flow

direction being used by the device. Flow direction is the direction in which the UI elements on the page are scanned by the eye. The enumeration values are:

- `LeftToRight`
- `RightToLeft`
- `MatchParent`

In XAML, the `Device.FlowDirection` value can be retrieved by using the `x:Static` markup extension:

```
<ContentPage ... FlowDirection="{x:Static Device.FlowDirection}"> />
```

The equivalent code in C# is:

```
this.FlowDirection = Device.FlowDirection;
```

For more information about flow direction, see [Right-to-left Localization](#).

## Device.Styles

The `Styles` property contains built-in style definitions that can be applied to some controls' (such as `Label`) `Style` property. The available styles are:

- `BodyStyle`
- `CaptionStyle`
- `ListItemDetailTextStyle`
- `ListItemTextStyle`
- `SubtitleStyle`
- `TitleStyle`

## Device.GetNamedSize

`GetNamedSize` can be used when setting `FontSize` in C# code:

```
myLabel.FontSize = Device.GetNamedSize (NamedSize.Small, myLabel);
someLabel.FontSize = Device.OnPlatform (
    24,           // hardcoded size
    Device.GetNamedSize (NamedSize.Medium, someLabel),
    Device.GetNamedSize (NamedSize.Large, someLabel)
);
```

## Device.OpenUri

The `openUri` method can be used to trigger operations on the underlying platform, such as open a URL in the native web browser (**Safari** on iOS or **Internet** on Android).

```
Device.OpenUri(new Uri("https://evolve.xamarin.com/"));
```

The [WebView sample](#) includes an example using `openUri` to open URLs and also trigger phone calls.

The [Maps sample](#) also uses `Device.OpenUri` to display maps and directions using the native **Maps** apps on iOS and Android.

## Device.StartTimer

The `Device` class also has a `StartTimer` method which provides a simple way to trigger time-dependent tasks that works in Xamarin.Forms common code, including a .NET Standard library. Pass a `TimeSpan` to set the interval and return `true` to keep the timer running or `false` to stop it after the current invocation.

```
Device.StartTimer (new TimeSpan (0, 0, 60), () => {
    // do something every 60 seconds
    return true; // runs again, or false to stop
});
```

If the code inside the timer interacts with the user-interface (such as setting the text of a `Label` or displaying an alert) it should be done inside a `BeginInvokeOnMainThread` expression (see below).

## Device.BeginInvokeOnMainThread

User interface elements should never be accessed by background threads, such as code running in a timer or a completion handler for asynchronous operations like web requests. Any background code that needs to update the user interface should be wrapped inside `BeginInvokeOnMainThread`. This is the equivalent of `InvokeOnMainThread` on iOS, `RunOnUiThread` on Android, and `Dispatcher.RunAsync` on the Universal Windows Platform.

The Xamarin.Forms code is:

```
Device.BeginInvokeOnMainThread ( () => {
    // interact with UI elements
});
```

Note that methods using `async/await` do not need to use `BeginInvokeOnMainThread` if they are running from the main UI thread.

## Summary

The Xamarin.Forms `Device` class allows fine-grained control over functionality and layouts on a per-platform basis - even in common code (either .NET Standard library projects or Shared Projects).

## Related Links

- [Device Sample](#)
- [Styles Sample](#)
- [Device](#)

# iOS Platform Features

7/25/2018 • 2 minutes to read • [Edit Online](#)

## iOS-specific Formatting

Xamarin.Forms enables cross-platform user interface styles and colors to be set - but there are other options for setting the theme of your iOS using platform-specific APIs in the iOS project.

Read more about formatting the user interface using iOS-specific APIs, such as **Info.plist** configuration and the **UIAppearance** API.

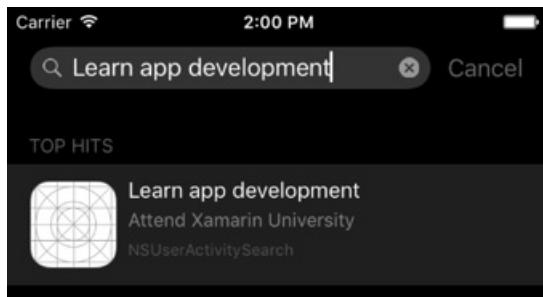


## iOS 9 Features

Using [custom renderers](#), the [DependencyService](#), and the [MessagingCenter](#), it's possible to incorporate a wide variety of native functionality into Xamarin.Forms apps for iOS.

The following recipes show how to incorporate iOS 9 features into the iOS part of a Xamarin.Forms app:

- [CoreSpotlight](#)
- [NSUserActivity](#)



# Adding iOS-specific Formatting

7/25/2018 • 2 minutes to read • [Edit Online](#)

One way to set iOS-specific formatting is to create a [custom renderer](#) for a control and set platform-specific styles and colors for each platform.

Other options to control the way your Xamarin.Forms iOS app's appearance include:

- Configuring display options in [Info.plist](#)
- Setting control styles via the [UIAppearance API](#)

These alternatives are discussed below.

## Customizing Info.plist

The [Info.plist](#) file lets you configure some aspects of an iOS application's rendering, such as how (and whether) the status bar is shown.

For example, the [Todo sample](#) uses the following code to set the navigation bar color and text color on all platforms:

```
var nav = new NavigationPage (new TodoListPage ());
nav.BarBackgroundColor = Color.FromHex("91CA47");
nav.BarTextColor = Color.White;
```

The result is shown in the screen snippet below. Notice that the status bar items are black (this cannot be set within Xamarin.Forms because it is a platform-specific feature).



Ideally the status bar would also be white - something we can accomplish directly in the iOS project. Add the following entries to the [Info.plist](#) to force the status bar to be white:

Status bar style	String	White
View controller-based status bar appearance	Boolean	No

or edit the corresponding [Info.plist](#) file directly to include:

```
<key>UIStatusBarStyle</key>
<string>UIStatusBarStyleLightContent</string>
<key>UIViewControllerBasedStatusBarAppearance</key>
<false/>
```

Now when the app is run, the navigation bar is green and its text is white (due to Xamarin.Forms formatting) *and* the status bar text is also white thanks to iOS-specific configuration:



buy apples ✓

## UIAppearance API

The `UIAppearance` API can be used to set visual properties on many iOS controls *without* having to create a custom renderer.

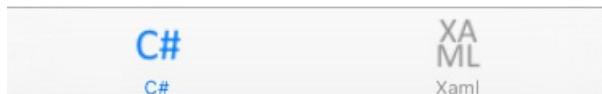
Adding a single line of code to the `AppDelegate.cs` `FinishedLaunching` method can style all controls of a given type using their `Appearance` property. The following code contains two examples - globally styling the tab bar and switch control:

### AppDelegate.cs in the iOS Project

```
public override bool FinishedLaunching (UIApplication app, NSDictionary options)
{
    // tab bar
    UITabBar.Appearance.SelectedImageTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
    // switch
    UISwitch.Appearance.OnTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
    // required Xamarin.Forms code
    Forms.Init ();
    LoadApplication (new App ());
    return base.FinishedLaunching (app, options);
}
```

### UITabBar

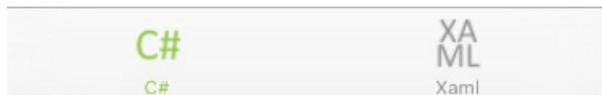
By default, the selected tab bar icon in a `TabbedPage` would be blue:



To change this behavior, set the `UITabBar.Appearance` property:

```
UITabBar.Appearance.SelectedImageTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
```

This causes the selected tab to be green:



Using this API lets you customize the appearance of the Xamarin.Forms `TabPage` on iOS with very little code. Refer to the [Customize Tabs recipe](#) for more details on using a custom renderer to set a specific font for the tab.

### UISwitch

The `Switch` control is another example that can be easily styled:

```
UISwitch.Appearance.OnTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
```

These two screen captures show the default `UISwitch` control on the left and the customized version (setting `Appearance`) on the right in the [Todo sample](#):

Done



Done



#### Other controls

Many iOS user interface controls can have their default colors and other attributes set using the [UIAppearance API](#).

## Related Links

- [UIAppearance](#)
- [Customize Tabs](#)

# GTK# Platform Setup

10/17/2018 • 5 minutes to read • [Edit Online](#)



Xamarin.Forms now has preview support for GTK# apps. GTK# is a graphical user interface toolkit that links the GTK+ toolkit and a variety of GNOME libraries, allowing the development of fully native GNOME graphics apps using Mono and .NET. This article demonstrates how to add a GTK# project to a Xamarin.Forms solution.

Before you start, create a new Xamarin.Forms solution, or use an existing Xamarin.Forms solution, for example, [GameOfLife](#).

## NOTE

While this article focuses on adding a GTK# app to a Xamarin.Forms solution in VS2017 and Visual Studio for Mac, it can also be performed in [MonoDevelop](#) for Linux.

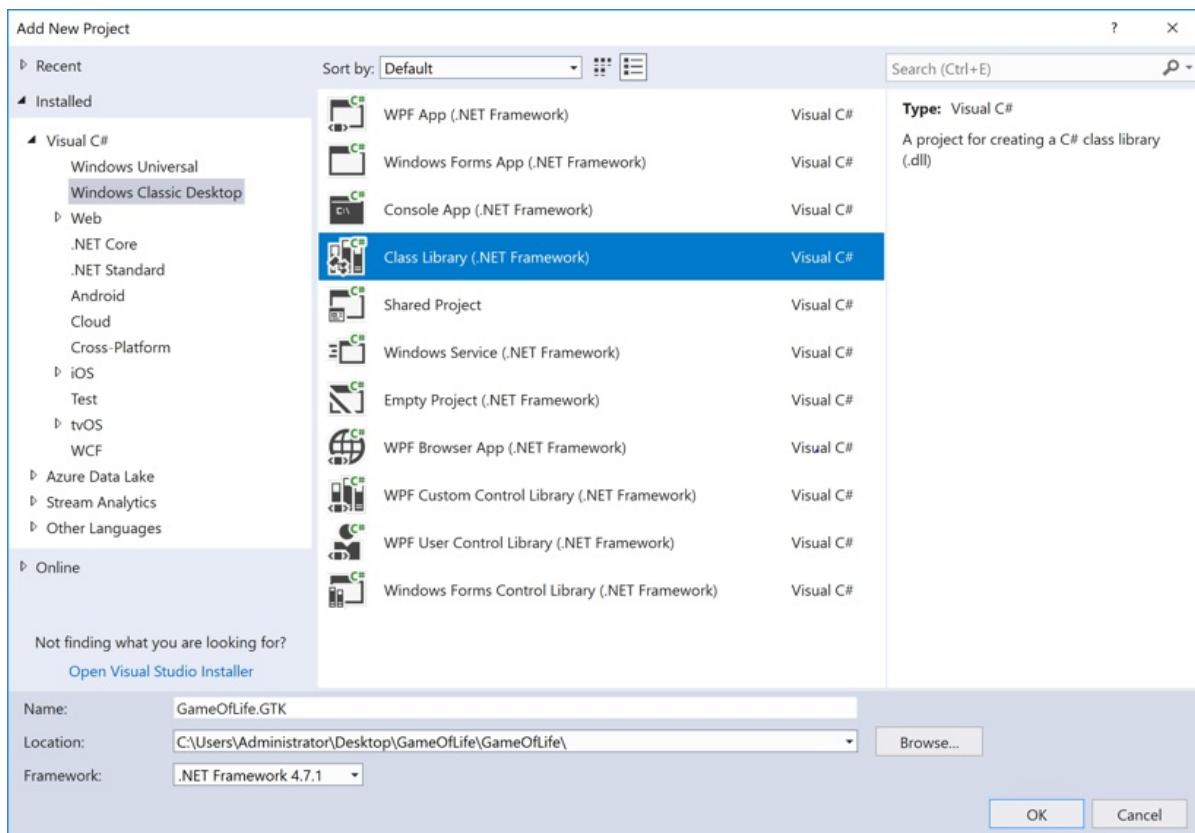
## Adding a GTK# App

GTK# for macOS and Linux is installed as part of [Mono](#). GTK# for .NET can be installed on Windows with the [GTK# Installer](#).

- [Visual Studio](#)
- [Visual Studio for Mac](#)

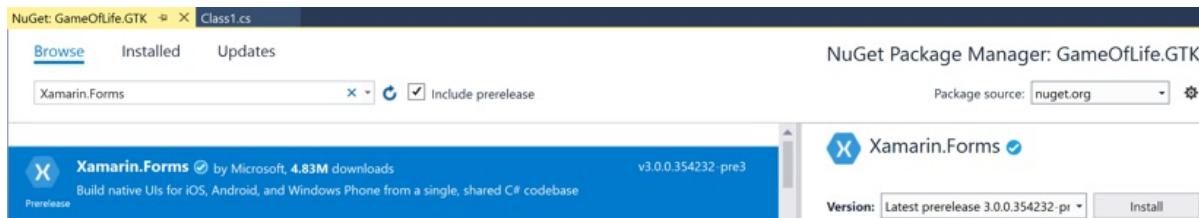
Follow these instructions to add a GTK# app that will run on the Windows desktop:

1. In Visual Studio 2017, right-click on the solution name in **Solution Explorer** and choose **Add > New Project....**
2. In the **New Project** window, at the left select **Visual C#** and **Windows Classic Desktop**. In the list of project types, choose **Class Library (.NET Framework)**, and ensure that the **Framework** drop-down is set to a minimum of .NET Framework 4.7.
3. Type a name for the project with a **GTK** extension, for example **GameOfLife.GTK**. Click the **Browse** button, select the folder containing the other platform projects, and press **Select Folder**. This will put the GTK project in the same directory as the other projects in the solution.



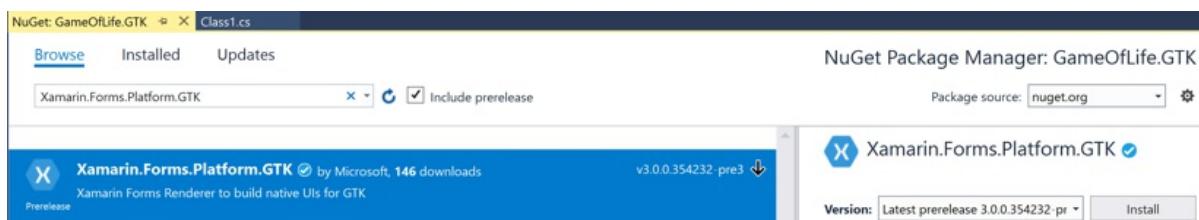
Press the **OK** button to create the project.

- In the **Solution Explorer**, right click the new GTK project and select **Manage NuGet Packages**. Select the **Browse** tab, and search for **Xamarin.Forms** 3.0 or greater.



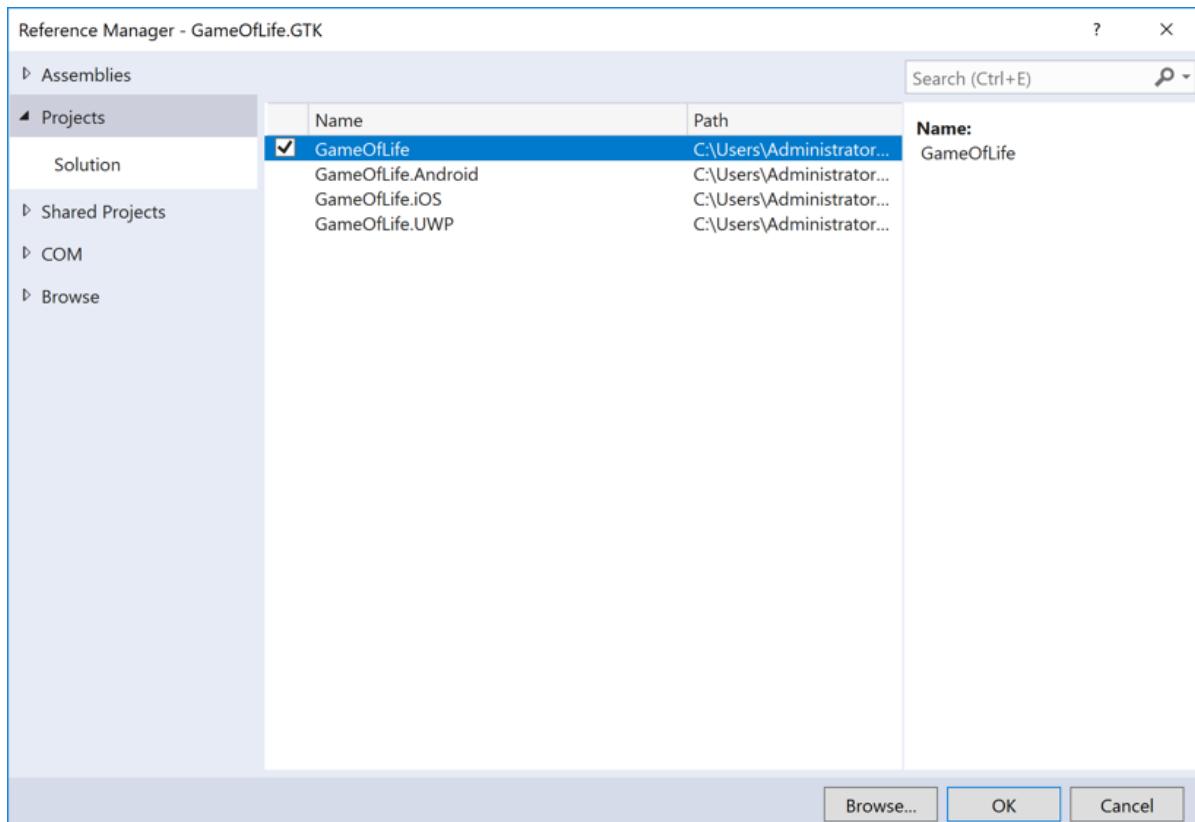
Select the package and click the **Install** button.

- Now search for the **Xamarin.Forms.Platform.GTK** 3.0 package or greater.

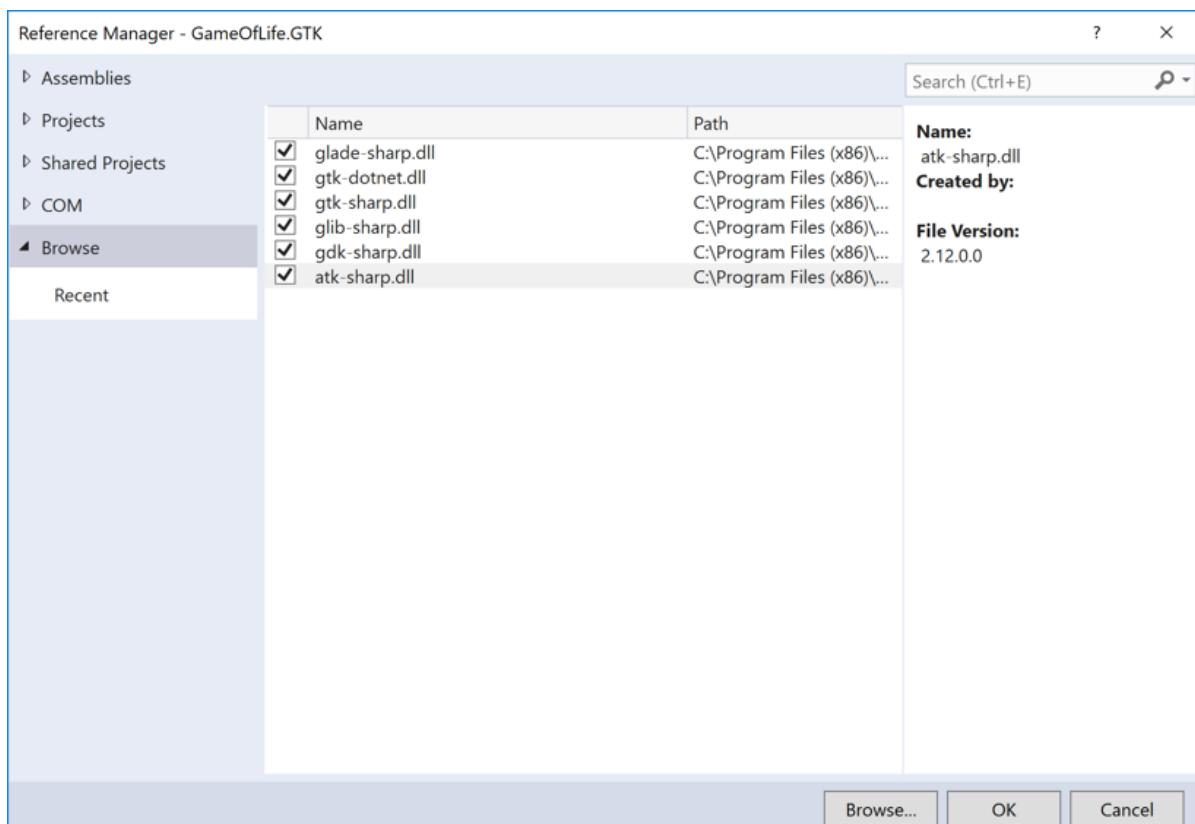


Select the package and click the **Install** button.

- In the **Solution Explorer**, right-click the solution name and select **Manage NuGet Packages for Solution**. Select the **Update** tab and the **Xamarin.Forms** package. Select all the projects and update them to the same Xamarin.Forms version as used by the GTK project.
- In the **Solution Explorer**, right-click on **References** in the GTK project. In the **Reference Manager** dialog, select **Projects** at the left, and check the checkbox adjacent to the .NET Standard or Shared project:



8. In the **Reference Manager** dialog, press the **Browse** button and browse to the **C:\Program Files (x86)\GtkSharp\2.12\lib** folder and select the **atk-sharp.dll**, **gdk-sharp.dll**, **glade-sharp.dll**, **glib-sharp.dll**, **gtk-dotnet.dll**, **gtk-sharp.dll** files.



Press the **OK** button to add the references.

9. In the GTK project, rename **Class1.cs** to **Program.cs**.
10. In the GTK project, edit the **Program.cs** file so that it resembles the following code:

```

using System;
using Xamarin.Forms;
using Xamarin.Forms.Platform.GTK;

namespace GameOfLife.GTK
{
    class MainClass
    {
        [STAThread]
        public static void Main(string[] args)
        {
            Gtk.Application.Init();
            Forms.Init();

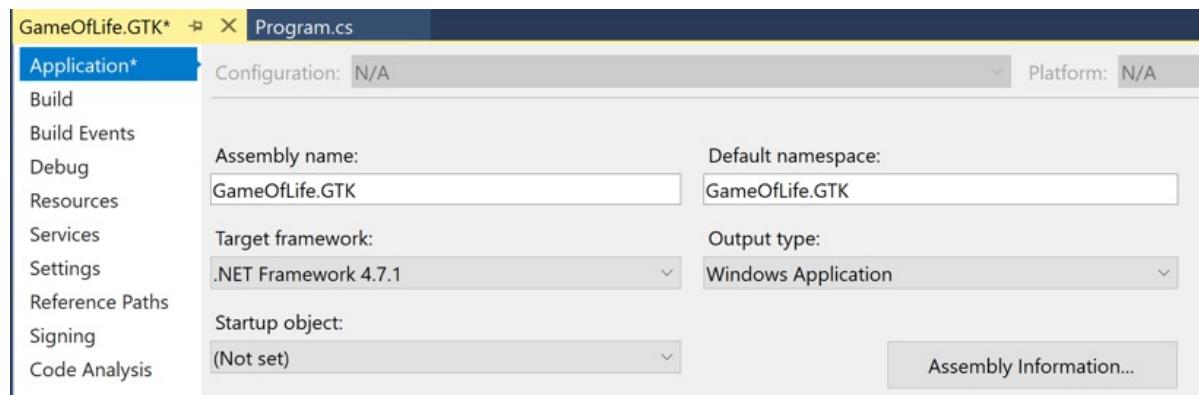
            var app = new App();
            var window = new FormsWindow();
            window.LoadApplication(app);
            window.SetApplicationTitle("Game of Life");
            window.Show();

            Gtk.Application.Run();
        }
    }
}

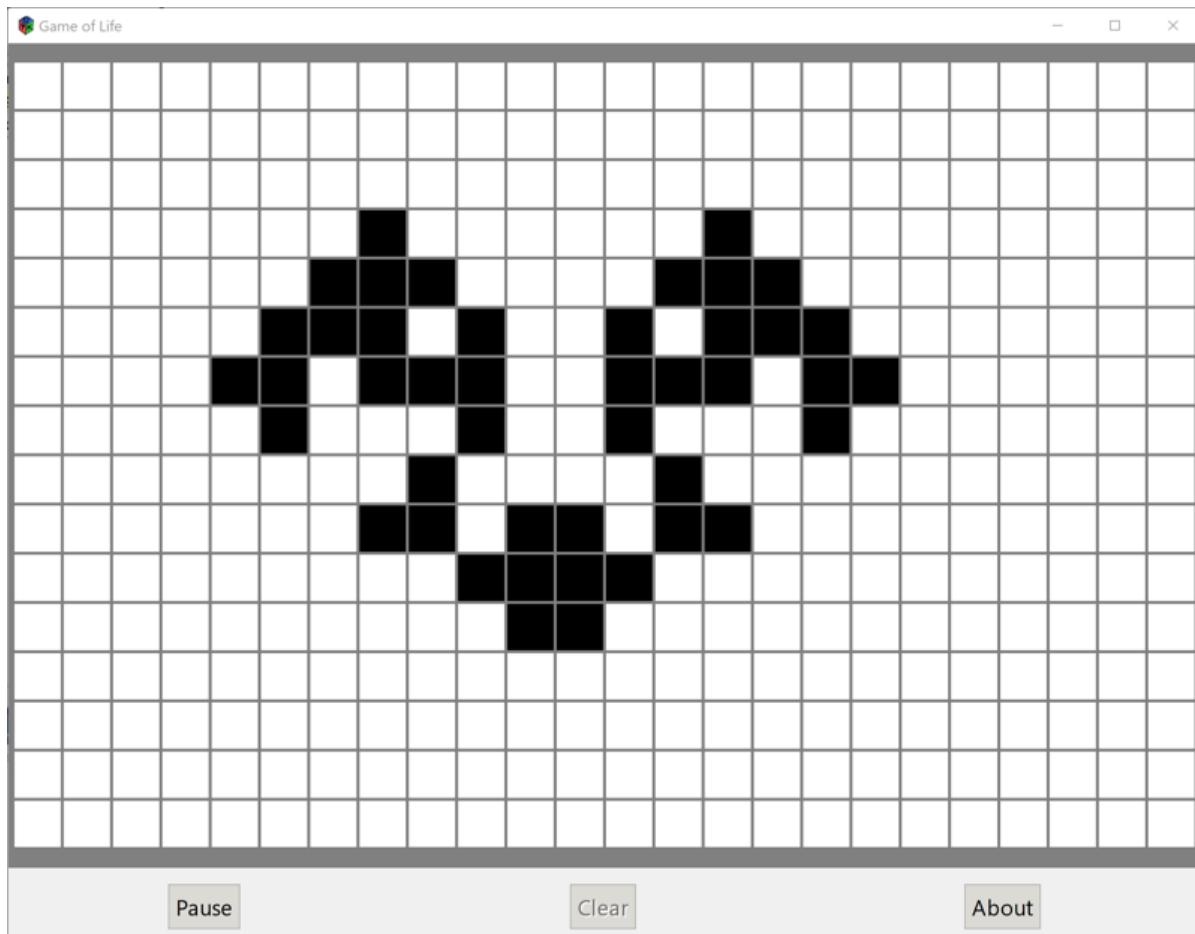
```

This code initializes GTK# and Xamarin.Forms, creates an application window, and runs the app.

11. In the **Solution Explorer**, right click the GTK project and select **Properties**.
12. In the **Properties** window, select the **Application** tab and change the **Output type** drop-down to **Windows Application**.



13. In the **Solution Explorer**, right-click the WPF project and select **Set as Startup Project**. Press F5 to run the program with the Visual Studio debugger on the Windows desktop:



## Next Steps

### Platform Specifics

You can determine what platform your Xamarin.Forms application is running on from either XAML or code. This allows you to change program characteristics when it's running on GTK#. In code, compare the value of `Device.RuntimePlatform` with the `Device.GTK` constant (which equals the string "GTK"). If there's a match, the application is running on GTK#.

In XAML, you can use the `OnPlatform` tag to select a property value specific to the platform:

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White" />
        <On Platform="macOS" Value="White" />
        <On Platform="Android" Value="Black" />
        <On Platform="GTK" Value="Blue" />
    </OnPlatform>
</Button.TextColor>
```

### Application Icon

You can set the app icon at startup:

```
window.SetApplicationIcon("icon.png");
```

### Themes

There are a wide variety of themes available for GTK#, and they can be used from a Xamarin.Forms app:

```
GtkThemes.Init ();
GtkThemes.LoadCustomTheme ("Themes/gtkrc");
```

## Native Forms

Native Forms allows Xamarin.Forms [ContentPage](#)-derived pages to be consumed by native projects, including GTK# projects. This can be accomplished by creating an instance of the [ContentPage](#)-derived page and converting it to the native GTK# type using the [CreateContainer](#) extension method:

```
var settingsView = new SettingsView().CreateContainer();
vbox.PackEnd(settingsView, true, true, 0);
```

For more information about Native Forms, see [Native Forms](#).

## Issues

This is a Preview, so you should expect that not everything is production ready. For the current implementation status, see [Status](#), and for the current known issues, see [Pending & Known Issues](#).

# Mac Platform Setup

11/13/2018 • 3 minutes to read • [Edit Online](#)



Before you start, create (or use an existing) Xamarin.Forms project. You can only add Mac apps using Visual Studio for Mac.

## **Adding a macOS project to Xamarin.Forms, by [Xamarin University](#)**

### Adding a Mac App

Follow these instructions to add a Mac app that will run on macOS Sierra and macOS El Capitan:

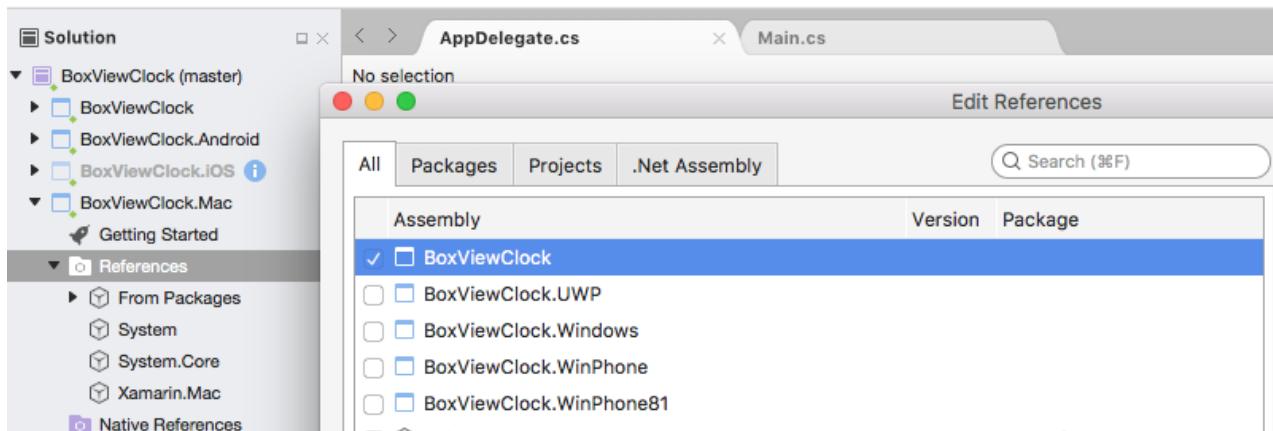
1. In Visual Studio for Mac, right-click on the existing Xamarin.Forms solution and choose **Add > Add New Project...**
2. In the **New Project** window choose **Mac > App > Cocoa App** and press **Next**.
3. Type an **App Name** (and optionally choose a different name for the Dock Item), then press **Next**.
4. Review the configuration and press **Create**. These steps are shown in below:

A screenshot of the Xamarin Studio interface. The top menu bar includes File, Edit, View, Search, Project, Build, Run, Version Control, Tools, Window, and Help. The title bar shows "Xamarin Studio" and the current project path: WeatherApp.iOS > Debug > Default. A status bar at the bottom indicates "Xamarin Studio Enterprise".  
  
The left side features the Solution Explorer, which lists the solution "WeatherApp (master)" with projects: WeatherApp, WeatherApp.Droid, WeatherApp.iOS (selected), WeatherApp.UWP, WeatherApp.Windows, and WeatherApp.WinPhone.  
  
The main workspace contains a code editor for "App.cs". The code is as follows:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 using Xamarin.Forms;
7
8 namespace WeatherApp
9 {
10     public class App : Application
11     {
12         public App()
13         {
14             MainPage = new NavigationPage(new WeatherPage());
15         }
16
17         protected override void OnStart()
18         {
19             // Handle when your app starts
20         }
21
22         protected override void OnSleep()
23         {
24             // Handle when your app sleeps
25         }
26
27         protected override void OnResume()
28     }
```

5. In the Mac project, right-click on **Packages > Add Packages...** to add the **Xamarin.Forms/2.3.5.235-pre2** NuGet. You should also update the other projects to this version.
6. In the Mac project, right-click on **References** and add a reference to the Xamarin.Forms project (either

Shared Project or .NET Standard library project).



7. Update `Main.cs` to initialize the `AppDelegate`:

```
static class MainClass
{
    static void Main(string[] args)
    {
        NSApplication.Init();
        NSApplication.SharedApplication.Delegate = new AppDelegate(); // add this line
        NSApplication.Main(args);
    }
}
```

8. Update `AppDelegate` to initialize `Xamarin.Forms`, create a window, and load the `Xamarin.Forms` application (remembering to set an appropriate `Title`). If you have other dependencies that need to be initialized, do that here as well.

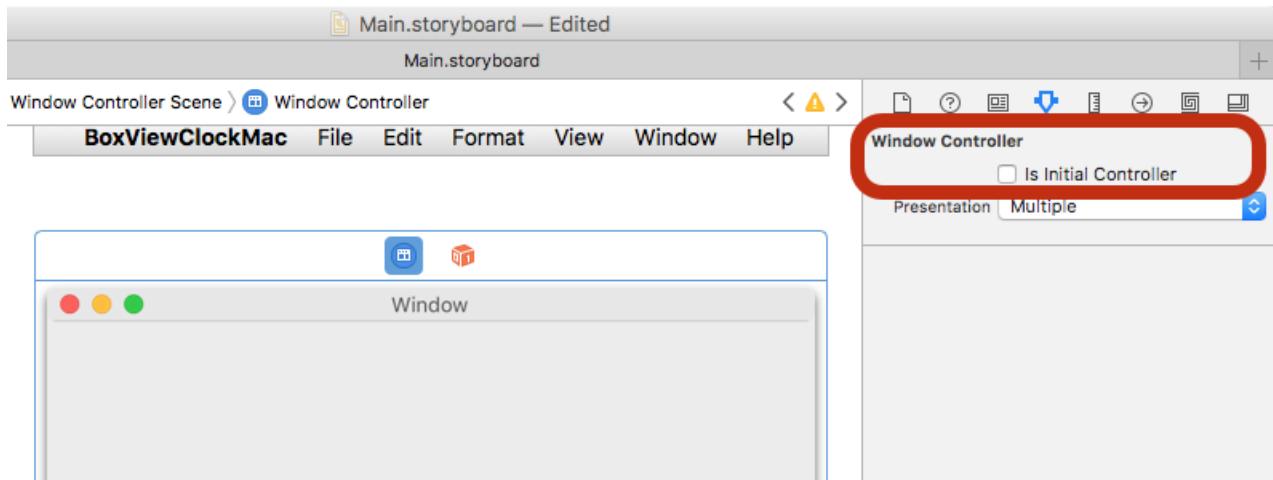
```
using Xamarin.Forms;
using Xamarin.Forms.Platform.MacOS;
// also add a using for the Xamarin.Forms project, if the namespace is different to this file
...
[Register("AppDelegate")]
public class AppDelegate : FormsApplicationDelegate
{
    NSWindow window;
    public AppDelegate()
    {
        var style = NSWindowStyle.Closable | NSWindowStyle.Resizable | NSWindowStyle.Titled;

        var rect = new CoreGraphics.CGRect(200, 1000, 1024, 768);
        window = new NSWindow(rect, style, NSBackingStore.Buffered, false);
        window.Title = "Xamarin.Forms on Mac!"; // choose your own Title here
        window.TitleVisibility = NSWindowTitleVisibility.Hidden;
    }

    public override NSWindow MainWindow
    {
        get { return window; }
    }

    public override void DidFinishLaunching(NSNotification notification)
    {
        Forms.Init();
        LoadApplication(new App());
        base.DidFinishLaunching(notification);
    }
}
```

9. Double-click **Main.storyboard** to edit in Xcode. Select the **Window** and *unchecked* the **Is Initial Controller** checkbox (this is because the code above creates a window):



You can edit the menu system in the storyboard to remove unwanted items.

10. Finally, add any local resources (eg. image files) from the existing platform projects that are required.
11. The Mac project should now run your Xamarin.Forms code on macOS!

## Next Steps

### Styling

With recent changes made to `OnPlatform` you can now target any number of platforms. That includes macOS.

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White"/>
        <On Platform="macOS" Value="White"/>
        <On Platform="Android" Value="Black"/>
    </OnPlatform>
</Button.TextColor>
```

Note you may also double up on platforms like this: `<On Platform="iOS, macOS" ...>`.

### Window Size and Position

You can adjust the initial size and location of the window in the `AppDelegate`:

```
var rect = new CoreGraphics.CGRect(200, 1000, 1024, 768); // x, y, width, height
```

## Known Issues

This is a Preview, so you should expect that not everything is production ready. Below are a few things you may encounter as you add macOS to your projects:

### Not all NuGets are ready for macOS

Packages must target "xamarinmac20" to work in a macOS project. You may find that some of the libraries you use do not yet support macOS.

In this case, you'll need to send a request to the project's maintainer to add it. Until they have support, you may need to look for alternatives.

### Missing Xamarin.Forms Features

Not all Xamarin.Forms features are complete in this preview; here is a list of some of the functionality that is not yet implemented:

- Footer
- Image – Aspect
- ListView – ScrollTo, UnevenRows support, refreshing, SeparatorColor, SeparatorVisibility
- MasterDetailPage – BackgroundColor
- Navigation – InsertPageBefore
- OpenGLRenderer
- Picker – Bindable/Observable implementation
- TabbedPage – BarBackgroundColor, BarTextColor
- TableView – UnevenRows
- ViewCell – IsEnabled, ForceUpdateSize
- WebView – most WebNavigationEvents

## Related Links

- [Xamarin.Mac](#)

# Xamarin.Forms in Xamarin Native Projects

11/12/2018 • 11 minutes to read • [Edit Online](#)

*Native Forms allow Xamarin.Forms ContentPage-derived pages to be consumed by native Xamarin.iOS, Xamarin.Android, and Universal Windows Platform (UWP) projects. Native projects can consume ContentPage-derived pages that are directly added to the project, or from a .NET Standard library, .NET Standard library, or Shared Project. This article explains how to consume ContentPage-derived pages that are directly added to native projects, and how to navigate between them.*

Typically, a Xamarin.Forms application includes one or more pages that derive from `ContentPage`, and these pages are shared by all platforms in a .NET Standard library project or Shared Project. However, Native Forms allows `ContentPage`-derived pages to be added directly to native Xamarin.iOS, Xamarin.Android, and UWP applications. Compared to having the native project consume `ContentPage`-derived pages from a .NET Standard library project or Shared Project, the advantage of adding pages directly to native projects is that the pages can be extended with native views. Native views can then be named in XAML with `x:Name` and referenced from the code-behind. For more information about native views, see [Native Views](#).

The process for consuming a Xamarin.Forms `ContentPage`-derived page in a native project is as follows:

1. Add the Xamarin.Forms NuGet package to the native project.
2. Add the `ContentPage`-derived page, and any dependencies, to the native project.
3. Call the `Forms.Init` method.
4. Construct an instance of the `ContentPage`-derived page and convert it to the appropriate native type using one of the following extension methods: `CreateViewController` for iOS, `CreateSupportFragment` for Android, or `CreateFrameworkElement` for UWP.
5. Navigate to the native type representation of the `ContentPage`-derived page using the native navigation API.

Xamarin.Forms must be initialized by calling the `Forms.Init` method before a native project can construct a `ContentPage`-derived page. Choosing when to do this primarily depends on when it's most convenient in your application flow – it could be performed at application startup, or just before the `ContentPage`-derived page is constructed. In this article, and the accompanying sample applications, the `Forms.Init` method is called at application startup.

## NOTE

The **NativeForms** sample application solution does not contain any Xamarin.Forms projects. Instead, it consists of a Xamarin.iOS project, a Xamarin.Android project, and a UWP project. Each project is a native project that uses Native Forms to consume `ContentPage`-derived pages. However, there's no reason why the native projects couldn't consume `ContentPage`-derived pages from a .NET Standard library project or Shared Project.

When using Native Forms, Xamarin.Forms features such as `DependencyService`, `MessagingCenter`, and the data binding engine, all still work. However, page navigation must be performed using the native navigation API.

## iOS

On iOS, the `FinishedLaunching` override in the `AppDelegate` class is typically the place to perform application startup related tasks. It's called after the application has launched, and is usually overridden to configure the main window and view controller. The following code example shows the `AppDelegate` class in the sample application:

```

[Register("AppDelegate")]
public class AppDelegate : UIApplicationDelegate
{
    public static AppDelegate Instance;

    UIWindow _window;
    UINavigationController _navigation;

    public override bool FinishedLaunching(UIApplication application, NSDictionary launchOptions)
    {
        Forms.Init();

        Instance = this;
        _window = new UIWindow(UIScreen.MainScreen.Bounds);

        UINavigationBar.AppearanceSetTitleTextAttributes(new UITextAttributes
        {
            TextColor = UIColor.Black
        });

        var mainPage = new PhonewordPage().CreateViewController();
        mainPage.Title = "Phoneword";

        _navigation = new UINavigationController(mainPage);
        _window.RootViewController = _navigation;
        _window.MakeKeyAndVisible();

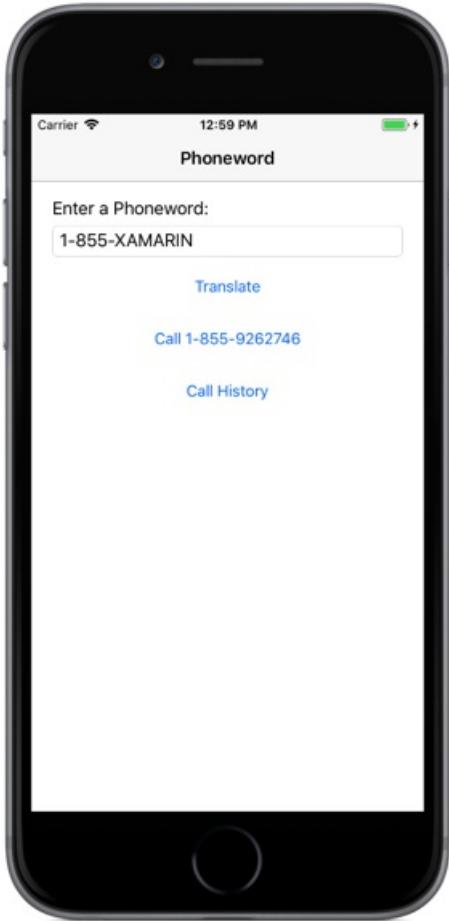
        return true;
    }
    ...
}

```

The `FinishedLaunching` method performs the following tasks:

- Xamarin.Forms is initialized by calling the `Forms.Init` method.
- A reference to the `AppDelegate` class is stored in the `static Instance` field. This is to provide a mechanism for other classes to call methods defined in the `AppDelegate` class.
- The `UIWindow`, which is the main container for views in native iOS applications, is created.
- The `PhonewordPage` class, which is a Xamarin.Forms `ContentPage`-derived page defined in XAML, is constructed and converted to a `UIViewController` using the `CreateViewController` extension method.
- The `Title` property of the `UIViewController` is set, which will be displayed on the `UINavigationBar`.
- A `UINavigationController` is created for managing hierarchical navigation. The `UINavigationController` class manages a stack of view controllers, and the `UIViewController` passed into the constructor will be presented initially when the `UINavigationController` is loaded.
- The `UINavigationController` instance is set as the top-level `UIViewController` for the `UIWindow`, and the `UIWindow` is set as the key window for the application and is made visible.

Once the `FinishedLaunching` method has executed, the UI defined in the Xamarin.Forms `PhonewordPage` class will be displayed, as shown in the following screenshot:



Interacting with the UI, for example by tapping on a `Button`, will result in event handlers in the `PhonewordPage` code-behind executing. For example, when a user taps the **Call History** button, the following event handler is executed:

```
void OnCallHistory(object sender, EventArgs e)
{
    AppDelegate.Instance.NavigateToCallHistoryPage();
}
```

The `static AppDelegate.Instance` field allows the `AppDelegate.NavigateToCallHistoryPage` method to be invoked, which is shown in the following code example:

```
public void NavigateToCallHistoryPage()
{
    var callHistoryPage = new CallHistoryPage().CreateViewController();
    callHistoryPage.Title = "Call History";
    _navigation.PushViewController(callHistoryPage, true);
}
```

The `NavigateToCallHistoryPage` method converts the `Xamarin.Forms ContentPage`-derived page to a `UIViewController` with the `CreateViewController` extension method, and sets the `Title` property of the `UIViewController`. The `UIViewController` is then pushed onto `UINavigationController` by the `PushViewController` method. Therefore, the UI defined in the `Xamarin.Forms CallHistoryPage` class will be displayed, as shown in the following screenshot:



When the `CallHistoryPage` is displayed, tapping the back arrow will pop the `UIViewController` for the `CallHistoryPage` class from the `UINavigationController`, returning the user to the `UIViewController` for the `PhonewordPage` class.

## Android

On Android, the `OnCreate` override in the `MainActivity` class is typically the place to perform application startup related tasks. The following code example shows the `MainActivity` class in the sample application:

```

public class MainActivity : AppCompatActivity
{
    public static MainActivity Instance;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        Forms.Init(this, bundle);
        Instance = this;

        SetContentView(Resource.Layout.Main);
        var toolbar = FindViewById<Toolbar>(Resource.Id.toolbar);
        SetSupportActionBar(toolbar);
        SupportActionBar.Title = "Phoneword";

        var mainPage = new PhonewordPage().CreateSupportFragment(this);
        SupportFragmentManager
            .BeginTransaction()
            .Replace(Resource.Id.fragment_frame_layout, mainPage)
            .Commit();
        ...
    }
    ...
}

```

The `OnCreate` method performs the following tasks:

- Xamarin.Forms is initialized by calling the `Forms.Init` method.
- A reference to the `MainActivity` class is stored in the `static Instance` field. This is to provide a mechanism for other classes to call methods defined in the `MainActivity` class.
- The `Activity` content is set from a layout resource. In the sample application, the layout consists of a `LinearLayout` that contains a `Toolbar`, and a `FrameLayout` to act as a fragment container.
- The `Toolbar` is retrieved and set as the action bar for the `Activity`, and the action bar title is set.
- The `PhonewordPage` class, which is a Xamarin.Forms `ContentPage`-derived page defined in XAML, is constructed and converted to a `Fragment` using the `CreateSupportFragment` extension method.
- The `SupportFragmentManager` class creates and commits a transaction that replaces the `FrameLayout` instance with the `Fragment` for the `PhonewordPage` class.

For more information about Fragments, see [Fragments](#).

Once the `OnCreate` method has executed, the UI defined in the Xamarin.Forms `PhonewordPage` class will be displayed, as shown in the following screenshot:



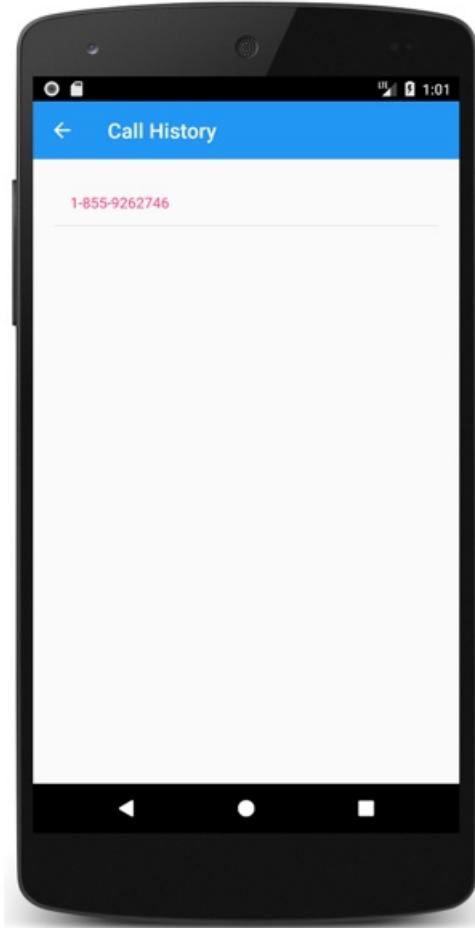
Interacting with the UI, for example by tapping on a `Button`, will result in event handlers in the `PhonewordPage` code-behind executing. For example, when a user taps the **Call History** button, the following event handler is executed:

```
void OnCallHistory(object sender, EventArgs e)
{
    MainActivity.Instance.NavigateToCallHistoryPage();
}
```

The `static MainActivity.Instance` field allows the `MainActivity.NavigateToCallHistoryPage` method to be invoked, which is shown in the following code example:

```
public void NavigateToCallHistoryPage()
{
    var callHistoryPage = new CallHistoryPage().CreateSupportFragment(this);
    SupportFragmentManager
        .BeginTransaction()
        .AddToBackStack(null)
        .Replace(Resource.Id.fragment_frame_layout, callHistoryPage)
        .Commit();
}
```

The `NavigateToCallHistoryPage` method converts the Xamarin.Forms `ContentPage`-derived page to a `Fragment` with the `CreateSupportFragment` extension method, and adds the `Fragment` to the fragment back stack. Therefore, the UI defined in the Xamarin.Forms `CallHistoryPage` will be displayed, as shown in the following screenshot:



When the `CallHistoryPage` is displayed, tapping the back arrow will pop the `Fragment` for the `CallHistoryPage` from the fragment back stack, returning the user to the `Fragment` for the `PhonewordPage` class.

### Enabling Back Navigation Support

The `SupportFragmentManager` class has a `BackStackChanged` event that fires whenever the content of the fragment back stack changes. The `OnCreate` method in the `MainActivity` class contains an anonymous event handler for this event:

```
SupportFragmentManager.BackStackChanged += (sender, e) =>
{
    bool hasBack = SupportFragmentManager.BackStackEntryCount > 0;
    SupportActionBar.SetHomeButtonEnabled(hasBack);
    SupportActionBar.SetDisplayHomeAsUpEnabled(hasBack);
    SupportActionBar.Title = hasBack ? "Call History" : "Phoneword";
};
```

This event handler displays a back button on the action bar provided that there's one or more `Fragment` instances on the fragment back stack. The response to tapping the back button is handled by the `OnOptionsItemSelected` override:

```
public override bool OnOptionsItemSelected(Android.Views.IMenuItem item)
{
    if (item.ItemId == global::Android.Resource.Id.Home && SupportFragmentManager.BackStackEntryCount > 0)
    {
        SupportFragmentManager.PopBackStack();
        return true;
    }
    return base.OnOptionsItemSelected(item);
}
```

The `OnOptionsItemSelected` override is called whenever an item in the options menu is selected. This implementation pops the current fragment from the fragment back stack, provided that the back button has been selected and there are one or more `Fragment` instances on the fragment back stack.

## Multiple Activities

When an application is composed of multiple activities, `ContentPage`-derived pages can be embedded into each of the activities. In this scenario, the `Forms.Init` method need be called only in the `OnCreate` override of the first `Activity` that embeds a Xamarin.Forms `ContentPage`. However, this has the following impact:

- The value of `Xamarin.Forms.Color.Accent` will be taken from the `Activity` that called the `Forms.Init` method.
- The value of `Xamarin.Forms.Application.Current` will be associated with the `Activity` that called the `Forms.Init` method.

## Choosing a File

When embedding a `ContentPage`-derived page that uses a `WebView` that needs to support an HTML "Choose File" button, the `Activity` will need to override the `OnActivityResult` method:

```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);
    ActivityResultCallbackRegistry.InvokeCallback(requestCode, resultCode, data);
}
```

## UWP

On UWP, the native `App` class is typically the place to perform application startup related tasks. Xamarin.Forms is usually initialized, in Xamarin.Forms UWP applications, in the `OnLaunched` override in the native `App` class, to pass the `LaunchActivatedEventArgs` argument to the `Forms.Init` method. For this reason, native UWP applications that consume a Xamarin.Forms `ContentPage`-derived page can most easily call the `Forms.Init` method from the `App.OnLaunched` method.

By default, the native `App` class launches the `MainPage` class as the first page of the application. The following code example shows the `MainPage` class in the sample application:

```
public sealed partial class MainPage : Page
{
    public static MainPage Instance;

    public MainPage()
    {
        this.InitializeComponent();
        this.NavigationCacheMode = NavigationCacheMode.Enabled;
        Instance = this;
        this.Content = new Phoneword.UWP.Views.PhonewordPage().CreateFrameworkElement();
    }
    ...
}
```

The `MainPage` constructor performs the following tasks:

- Caching is enabled for the page, so that a new `MainPage` isn't constructed when a user navigates back to the page.
- A reference to the `MainPage` class is stored in the `static` `Instance` field. This is to provide a mechanism for other classes to call methods defined in the `MainPage` class.
- The `PhonewordPage` class, which is a Xamarin.Forms `ContentPage`-derived page defined in XAML, is

constructed and converted to a `FrameworkElement` using the `CreateFrameworkElement` extension method, and then set as the content of the `MainPage` class.

Once the `MainPage` constructor has executed, the UI defined in the `Xamarin.Forms PhonewordPage` class will be displayed, as shown in the following screenshot:



Interacting with the UI, for example by tapping on a `Button`, will result in event handlers in the `PhonewordPage` code-behind executing. For example, when a user taps the **Call History** button, the following event handler is executed:

```
void OnCallHistory(object sender, EventArgs e)
{
    Phoneword.UWP.MainPage.Instance.NavigateToCallHistoryPage();
}
```

The `static MainPage.Instance` field allows the `MainPage.NavigateToCallHistoryPage` method to be invoked, which is shown in the following code example:

```
public void NavigateToCallHistoryPage()
{
    this.Frame.Navigate(new CallHistoryPage());
}
```

Navigation in UWP is typically performed with the `Frame.Navigate` method, which takes a `Page` argument. `Xamarin.Forms` defines a `Frame.Navigate` extension method that takes a `ContentPage`-derived page instance. Therefore, when the `NavigateToCallHistoryPage` method executes, the UI defined in the `Xamarin.Forms CallHistoryPage` will be displayed, as shown in the following screenshot:



When the `CallHistoryPage` is displayed, tapping the back arrow will pop the `FrameworkElement` for the `CallHistoryPage` from the in-app back stack, returning the user to the `FrameworkElement` for the `PhonewordPage` class.

### Enabling Back Navigation Support

On UWP, applications must enable back navigation for all hardware and software back buttons, across different device form factors. This can be accomplished by registering an event handler for the `BackRequested` event, which can be performed in the `OnLaunched` method in the native `App` class:

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;

    if (rootFrame == null)
    {
        ...
        // Place the frame in the current Window
        Window.Current.Content = rootFrame;

        SystemNavigationManager.GetForCurrentView().BackRequested += OnBackRequested;
    }
    ...
}
```

When the application is launched, the `GetForCurrentView` method retrieves the `SystemNavigationManager` object associated with the current view, then registers an event handler for the `BackRequested` event. The application only receives this event if it's the foreground application, and in response, calls the `OnBackRequested` event handler:

```
void OnBackRequested(object sender, BackRequestedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame.CanGoBack)
    {
        e.Handled = true;
        rootFrame.GoBack();
    }
}
```

The `OnBackRequested` event handler calls the `GoBack` method on the root frame of the application and sets the `BackRequestedEventArgs.Handled` property to `true` to mark the event as handled. Failure to mark the event as handled could result in the system navigating away from the application (on the mobile device family) or ignoring the event (on the desktop device family).

The application relies on a system provided back button on a phone, but chooses whether to show a back button on the title bar on desktop devices. This is achieved by setting the `AppView backButtonVisibility` property to one of the `AppView backButtonVisibility` enumeration values:

```
void OnNavigated(object sender, NavigationEventArgs e)
{
    SystemNavigationManager.GetForCurrentView().AppView backButtonVisibility =
        ((Frame)sender).CanGoBack ? AppView backButtonVisibility.Visible :
    AppView backButtonVisibility.Collapsed;
}
```

The `OnNavigated` event handler, which is executed in response to the `Navigated` event firing, updates the visibility of the title bar back button when page navigation occurs. This ensures that the title bar back button is visible if the in-app back stack is not empty, or removed from the title bar if the in-app back stack is empty.

For more information about back navigation support on UWP, see [Navigation history and backwards navigation for UWP apps](#).

## Summary

Native Forms allow Xamarin.Forms `ContentPage`-derived pages to be consumed by native Xamarin.iOS, Xamarin.Android, and Universal Windows Platform (UWP) projects. Native projects can consume `ContentPage`-derived pages that are directly added to the project, or from a .NET Standard library project or Shared Project. This article explained how to consume `ContentPage`-derived pages that are directly added to native projects, and how to navigate between them.

## Related Links

- [NativeForms \(sample\)](#)
- [Native Views](#)

# Native Views in Xamarin.Forms

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Native views from iOS, Android, and the Universal Windows Platform (UWP) can be directly referenced from Xamarin.Forms. Properties and event handlers can be set on native views, and they can interact with Xamarin.Forms views.*

## Native Views in XAML

Native views from iOS, Android, and UWP can be directly referenced from Xamarin.Forms pages created using XAML.

## Native Views in C#

Native views from iOS, Android, and UWP can be directly referenced from Xamarin.Forms pages created using C#.

## Related Links

- [Native Forms](#)

# Native Views in XAML

7/12/2018 • 13 minutes to read • [Edit Online](#)

*Native views from iOS, Android, and the Universal Windows Platform can be directly referenced from Xamarin.Forms XAML files. Properties and event handlers can be set on native views, and they can interact with Xamarin.Forms views. This article demonstrates how to consume native views from Xamarin.Forms XAML files.*

This article discusses the following topics:

- [Consuming native views](#) – the process for consuming a native view from XAML.
- [Using native bindings](#) – data binding to and from properties of native views.
- [Passing arguments to native views](#) – passing arguments to native view constructors, and calling native view factory methods.
- [Referring to native views from code](#) – retrieving native view instances declared in a XAML file, from its code-behind file.
- [Subclassing native views](#) – subclassing native views to define a XAML-friendly API.

## Overview

To embed a native view into a Xamarin.Forms XAML file:

1. Add an `xmlns` namespace declaration in the XAML file for the namespace that contains the native view.
2. Create an instance of the native view in the XAML file.

### NOTE

XAMLC must be turned off for any XAML pages that use native views.

To reference a native view from a code-behind file, you must use a Shared Asset Project (SAP) and wrap the platform-specific code with conditional compilation directives. For more information see [Referring to Native Views from Code](#).

## Consuming Native Views

The following code example demonstrates consuming native views for each platform to a Xamarin.Forms

`ContentPage` :

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
    xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidLocal="clr-
    namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
    xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
        Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    x:Class="NativeViews.NativeViewDemo">
    <StackLayout Margin="20">
        <ios:UILabel Text="Hello World" TextColor="{x:Static ios:UIColor.Red}" View.HorizontalOptions="Start"
    />
        <androidWidget:TextView Text="Hello World" x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
    />
        <win:TextBlock Text="Hello World" />
    </StackLayout>
</ContentPage>

```

As well as specifying the `clr-namespace` and `assembly` for a native view namespace, a `targetPlatform` must also be specified. This should be set to one of values of the `TargetPlatform` enumeration, and will typically be set to `iOS`, `Android`, or `Windows`. At runtime, the XAML parser will ignore any XML namespace prefixes that have a `targetPlatform` that doesn't match the platform on which the application is running.

Each namespace declaration can be used to reference any class or structure from the specified namespace. For example, the `ios` namespace declaration can be used to reference any class or structure from the iOS `UIKit` namespace. Properties of the native view can be set through XAML, but the property and object types must match. For example, the `UILabel.TextColor` property is set to `UIColor.Red` using the `x:Static` markup extension and the `ios` namespace.

Bindable properties and attached bindable properties can also be set on native views by using the `Class.BindableProperty="value"` syntax. Each native view is wrapped in a platform-specific `NativeViewWrapper` instance, which derives from the `Xamarin.Forms.View` class. Setting a bindable property or attached bindable property on a native view transfers the property value to the wrapper. For example, a centered horizontal layout can be specified by setting `View.HorizontalOptions="Center"` on the native view.

#### **NOTE**

Note that styles can't be used with native views, because styles can only target properties that are backed by `BindableProperty` objects.

Android widget constructors generally require the Android `Context` object as an argument, and this can be made available through a static property in the `MainActivity` class. Therefore, when creating an Android widget in XAML, the `Context` object must generally be passed to the widget's constructor using the `x:Arguments` attribute with a `x:Static` markup extension. For more information, see [Passing Arguments to Native Views](#).

#### **NOTE**

Note that naming a native view with `x:Name` is not possible in either a .NET Standard library project or a Shared Asset Project (SAP). Doing so will generate a variable of the native type, which will cause a compilation error. However, native views can be wrapped in `ContentView` instances and retrieved in the code-behind file, provided that a SAP is being used. For more information, see [Referring to a Native View from Code](#).

## Native Bindings

Data binding is used to synchronize a UI with its data source, and simplifies how a Xamarin.Forms application

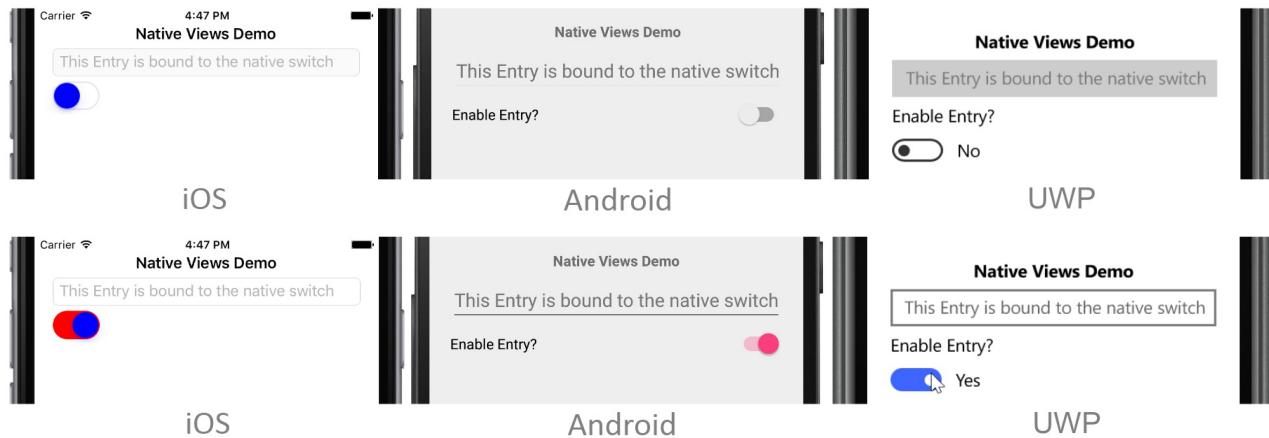
displays and interacts with its data. Provided that the source object implements the `INotifyPropertyChanged` interface, changes in the *source* object are automatically pushed to the *target* object by the binding framework, and changes in the *target* object can optionally be pushed to the *source* object.

Properties of native views can also use data binding. The following code example demonstrates data binding using properties of native views:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
    xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidLocal="clr-
    namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
    xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
        Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:local="clr-namespace:NativeSwitch"
    x:Class="NativeSwitch.NativeSwitchPage">
<StackLayout Margin="20">
    <Label Text="Native Views Demo" FontAttributes="Bold" HorizontalOptions="Center" />
    <Entry Placeholder="This Entry is bound to the native switch" IsEnabled="{Binding IsSwitchOn}" />
    <ios:UISwitch On="'{Binding Path=IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=ValueChanged}'"
        OnTintColor="{x:Static ios:UIColor.Red}"
        ThumbTintColor="{x:Static ios:UIColor.Blue}" />
    <androidWidget:Switch x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
        Checked="{Binding Path=IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=CheckedChange}"
        Text="Enable Entry?" />
    <win:ToggleSwitch Header="Enable Entry?">
        OffContent="No"
        OnContent="Yes"
        IsOn="{Binding IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=Toggled}" />
    </StackLayout>
</ContentPage>
```

The page contains an `Entry` whose `.IsEnabled` property binds to the `NativeSwitchPageViewModel.IsSwitchOn` property. The `BindingContext` of the page is set to a new instance of the `NativeSwitchPageViewModel` class in the code-behind file, with the `ViewModel` class implementing the `INotifyPropertyChanged` interface.

The page also contains a native switch for each platform. Each native switch uses a `TwoWay` binding to update the value of the `NativeSwitchPageViewModel.IsSwitchOn` property. Therefore, when the switch is off, the `Entry` is disabled, and when the switch is on, the `Entry` is enabled. The following screenshots show this functionality on each platform:



Two-way bindings are automatically supported provided that the native property implements `INotifyPropertyChanged`, or supports Key-Value Observing (KVO) on iOS, or is a `DependencyProperty` on UWP. However, many native views don't support property change notification. For these views, you can specify an `UpdateSourceEventName` property value as part of the binding expression. This property should be set to the name

of an event in the native view that signals when the target property has changed. Then, when the value of the native switch changes, the `Binding` class is notified that the user has changed the switch value, and the `NativeSwitchPageViewModel.IsSwitchOn` property value is updated.

## Passing Arguments to Native Views

Constructor arguments can be passed to native views using the `x:Arguments` attribute with a `x:static` markup extension. In addition, native view factory methods (`public static`) methods that return objects or values of the same type as the class or structure that defines the methods) can be called by specifying the method's name using the `x:FactoryMethod` attribute, and its arguments using the `x:Arguments` attribute.

The following code example demonstrates both techniques:

```

<ContentPage ...>
    xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
    xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidGraphics="clr-namespace:Android.Graphics;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidLocal="clr-
        namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
    xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
    Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:winMedia="clr-namespace:Windows.UI.Xaml.Media;assembly=Windows, Version=255.255.255.255,
    Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:winText="clr-namespace:Windows.UI.Text;assembly=Windows, Version=255.255.255.255,
    Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:winui="clr-namespace:Windows.UI;assembly=Windows, Version=255.255.255.255, Culture=neutral,
    PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows">
    ...
    <ios:UILabel Text="Simple Native Color Picker" View.HorizontalOptions="Center">
        <ios:UILabel.Font>
            <ios:UIFont x:FactoryMethod="FromName">
                <x:Arguments>
                    <x:String>Papyrus</x:String>
                    <x:Single>24</x:Single>
                </x:Arguments>
            </ios:UIFont>
        </ios:UILabel.Font>
    </ios:UILabel>
    <androidWidget:TextView x:Arguments="{x:Static androidLocal:MainActivity.Instance}">
        Text="Simple Native Color Picker"
        TextSize="24"
        View.HorizontalOptions="Center">
        <androidWidget:TextView.Typeface>
            <androidGraphics:Typeface x:FactoryMethod="Create">
                <x:Arguments>
                    <x:String>cursive</x:String>
                    <androidGraphics:TypefaceStyle>Normal</androidGraphics:TypefaceStyle>
                </x:Arguments>
            </androidGraphics:Typeface>
        </androidWidget:TextView.Typeface>
    </androidWidget:TextView>
    <winControls:TextBlock Text="Simple Native Color Picker"
        FontSize="20"
        FontStyle="{x:Static winText:FontStyle.Italic}"
        View.HorizontalOptions="Center">
        <winControls:TextBlock.FontFamily>
            <winMedia:FontFamily>
                <x:Arguments>
                    <x:String>Georgia</x:String>
                </x:Arguments>
            </winMedia:FontFamily>
        </winControls:TextBlock.FontFamily>
    </winControls:TextBlock>
    ...
</ContentPage>

```

The `UIFont.FromName` factory method is used to set the `UILabel.Font` property to a new `UIFont` on iOS. The `UIFont` name and size are specified by the method arguments that are children of the `x:Arguments` attribute.

The `Typeface.Create` factory method is used to set the `TextView.Typeface` property to a new `Typeface` on Android. The `Typeface` family name and style are specified by the method arguments that are children of the `x:Arguments` attribute.

The `FontFamily` constructor is used to set the `TextBlock.FontFamily` property to a new `FontFamily` on the Universal Windows Platform (UWP). The `FontFamily` name is specified by the method argument that is a child of the `x:Arguments` attribute.

## NOTE

Arguments must match the types required by the constructor or factory method.

The following screenshots show the result of specifying factory method and constructor arguments to set the font on different native views:



For more information about passing arguments in XAML, see [Passing Arguments in XAML](#).

## Referring to Native Views from Code

Although it's not possible to name a native view with the `x:Name` attribute, it is possible to retrieve a native view instance declared in a XAML file from its code-behind file in a Shared Access Project, provided that the native view is a child of a `ContentView` that specifies an `x:Name` attribute value. Then, inside conditional compilation directives in the code-behind file you should:

1. Retrieve the `ContentView.Content` property value and cast it to a platform-specific `NativeViewWrapper` type.
2. Retrieve the `NativeViewWrapper.NativeElement` property and cast it to the native view type.

The native API can then be invoked on the native view to perform the desired operations. This approach also offers the benefit that multiple XAML native views for different platforms can be children of the same `ContentView`. The following code example demonstrates this technique:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
    xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
    xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
        Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:local="clr-namespace:NativeViewInsideContentView"
    x:Class="NativeViewInsideContentView.NativeViewInsideContentViewPage">
    <StackLayout Margin="20">
        <ContentView x:Name="contentViewTextParent" HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand">
            <ios:UILabel Text="Text in a UILabel" TextColor="{x:Static ios:UIColor.Red}" />
            <androidWidget:TextView x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
                Text="Text in a TextView" />
            <winControls:TextBlock Text="Text in a TextBlock" />
        </ContentView>
        <ContentView x:Name="contentViewButtonParent" HorizontalOptions="Center"
            VerticalOptions="EndAndExpand">
            <ios:UIButton TouchUpInside="OnButtonTap" View.HorizontalOptions="Center"
                View.VerticalOptions="Center" />
            <androidWidget:Button x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
                Text="Scale and Rotate Text"
                Click="OnButtonTap" />
            <winControls:Button Content="Scale and Rotate Text" />
        </ContentView>
    </StackLayout>
</ContentPage>
```

In the example above, the native views for each platform are children of `ContentView` controls, with the `x:Name`

attribute value being used to retrieve the `ContentView` in the code-behind:

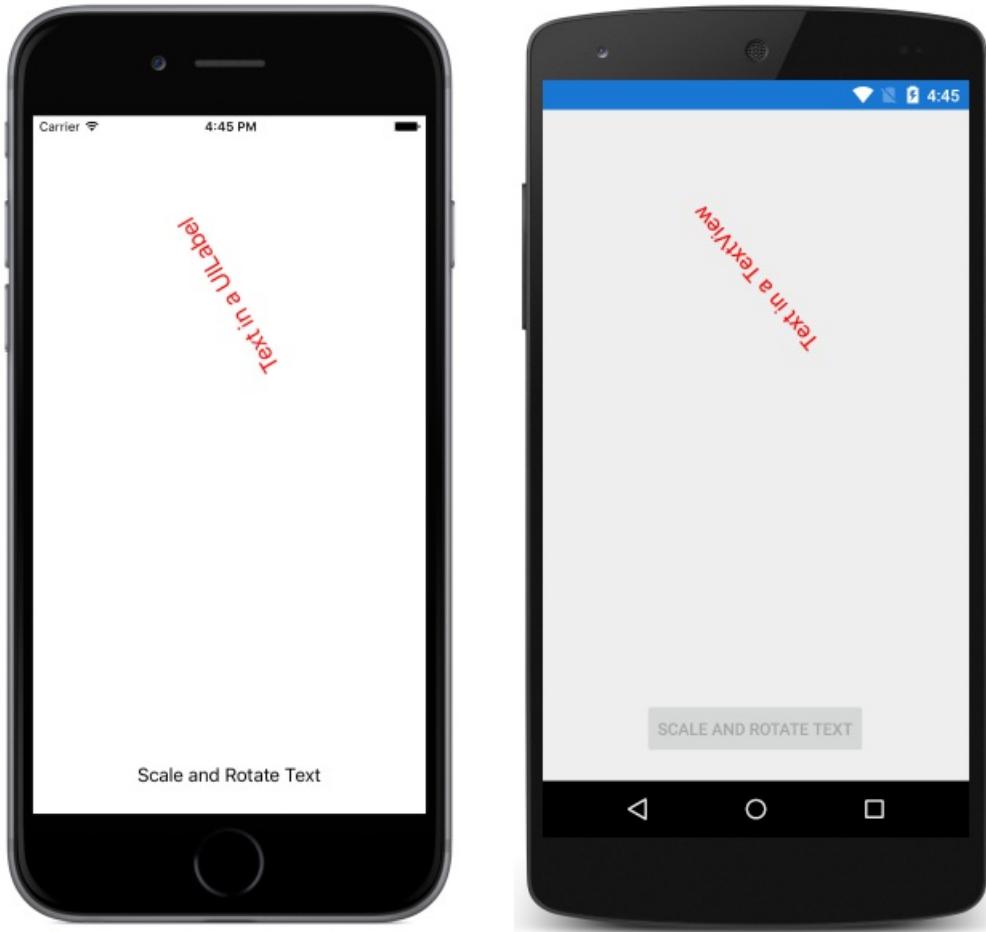
```
public partial class NativeViewInsideContentViewPage : ContentPage
{
    public NativeViewInsideContentViewPage()
    {
        InitializeComponent();

#if __IOS__
        var wrapper = (Xamarin.Forms.Platform.iOS.NativeViewWrapper)contentViewButtonParent.Content;
        var button = (UIKit.UITextField)wrapper.NativeView;
        button.SetTitle("Scale and Rotate Text", UIKit.UIControlState.Normal);
        button.SetTitleColor(UIKit.UIColor.Black, UIKit.UIControlState.Normal);
#endif
#if __ANDROID__
        var wrapper = (Xamarin.Forms.Platform.Android.NativeViewWrapper)contentViewTextParent.Content;
        var textView = (Android.Widget.TextView)wrapper.NativeView;
        textView.SetTextSize(Android.Graphics.TextSize.Large);
#endif
#if WINDOWS_UWP
        var textWrapper = (Xamarin.Forms.Platform.UWP.NativeViewWrapper)contentViewTextParent.Content;
        var textBlock = (Windows.UI.Xaml.Controls.TextBlock)textWrapper.NativeElement;
        textBlock.Foreground = new Windows.UI.Xaml.Media.SolidColorBrush(Windows.UI.Colors.Red);
        var buttonWrapper = (Xamarin.Forms.Platform.UWP.NativeViewWrapper)contentViewButtonParent.Content;
        var button = (Windows.UI.Xaml.Controls.Button)buttonWrapper.NativeElement;
        button.Click += (sender, args) => OnButtonTap(sender, EventArgs.Empty);
#endif
    }

    async void OnButtonTap(object sender, EventArgs e)
    {
        contentViewButtonParent.Content.IsEnabled = false;
        contentViewTextParent.Content.ScaleTo(2, 2000);
        await contentViewTextParent.Content.RotateTo(360, 2000);
        contentViewTextParent.Content.ScaleTo(1, 2000);
        await contentViewTextParent.Content.RelRotateTo(360, 2000);
        contentViewButtonParent.Content.IsEnabled = true;
    }
}
```

The `ContentView.Content` property is accessed to retrieve the wrapped native view as a platform-specific `NativeViewWrapper` instance. The `NativeViewWrapper.NativeElement` property is then accessed to retrieve the native view as its native type. The native view's API is then invoked to perform the desired operations.

The iOS and Android native buttons share the same `OnButtonTap` event handler, because each native button consumes an `EventHandler` delegate in response to a touch event. However, the Universal Windows Platform (UWP) uses a separate `RoutedEventHandler`, which in turn consumes the `OnButtonTap` event handler in this example. Therefore, when a native button is clicked, the `OnButtonTap` event handler executes, which scales and rotates the native control contained within the `ContentView` named `contentViewTextParent`. The following screenshots demonstrate this occurring on each platform:



## Subclassing Native Views

Many iOS and Android native views are not suitable for instantiating in XAML because they use methods, rather than properties, to set up the control. The solution to this issue is to subclass native views in wrappers that define a more XAML-friendly API that uses properties to setup the control, and that uses platform-independent events. The wrapped native views can then be placed in a Shared Asset Project (SAP) and surrounded with conditional compilation directives, or placed in platform-specific projects and referenced from XAML in a .NET Standard library project.

The following code example demonstrates a `Xamarin.Forms` page that consumes subclassed native views:

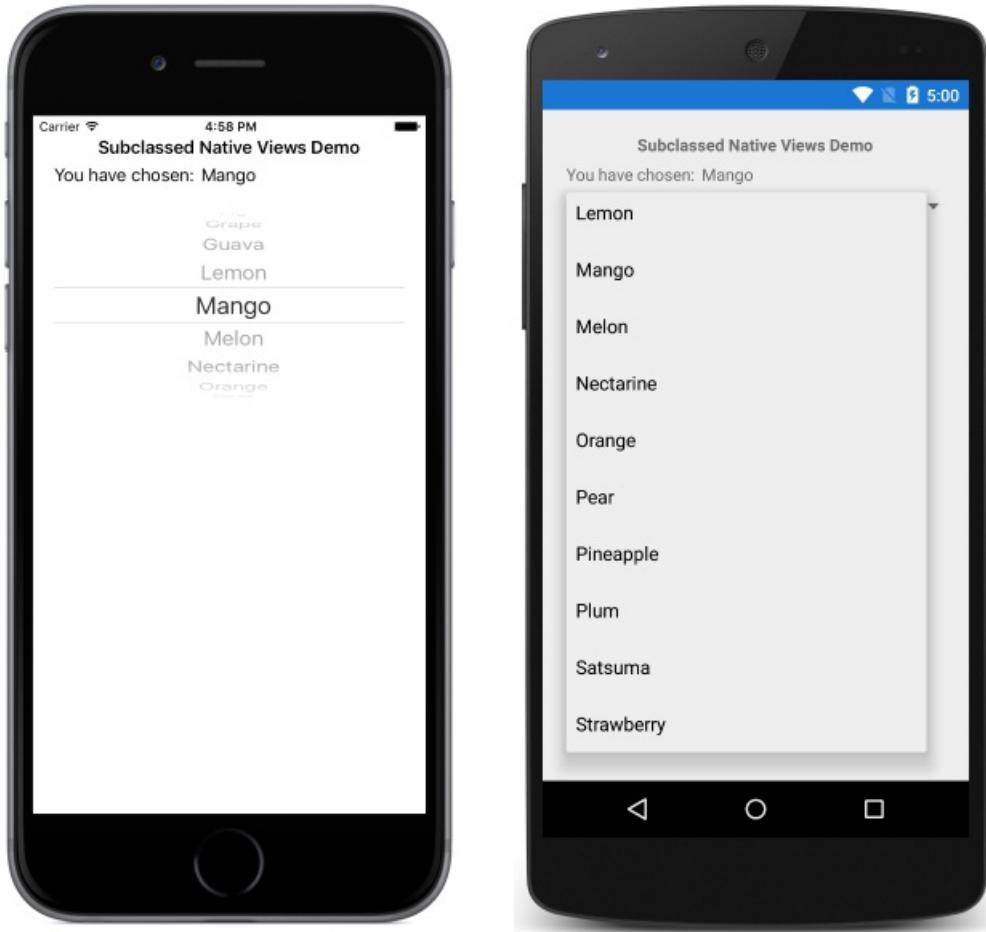
```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
    xmlns:iosLocal="clr-
    namespace:SubclassedNativeControls.iOS;assembly=SubclassedNativeControls.iOS;targetPlatform=iOS"
    xmlns:android="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
    xmlns:androidLocal="clr-
    namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
    xmlns:androidLocal="clr-
    namespace:SubclassedNativeControls.Droid;assembly=SubclassedNativeControls.Droid;targetPlatform=Android"
    xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
        Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
    xmlns:local="clr-namespace:SubclassedNativeControls"
    x:Class="SubclassedNativeControls.SubclassedNativeControlsPage">
<StackLayout Margin="20">
    <Label Text="Subclassed Native Views Demo" FontAttributes="Bold" HorizontalOptions="Center" />
    <StackLayout Orientation="Horizontal">
        <Label Text="You have chosen:" />
        <Label Text="{Binding SelectedFruit}" />
    </StackLayout>
    <iOSLocal:MyUIPickerView ItemsSource="{Binding Fruits}"
        SelectedItem="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=SelectedItemChanged}" />
    <androidLocal:MySpinner x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
        ItemsSource="{Binding Fruits}"
        SelectedObject="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=ItemSelected}" />
    <winControls:ComboBox ItemsSource="{Binding Fruits}"
        SelectedItem="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=SelectionChanged}" />
</StackLayout>
</ContentPage>

```

The page contains a `Label` that displays the fruit chosen by the user from a native control. The `Label` binds to the `SubclassedNativeControlsPageViewModel.SelectedFruit` property. The `BindingContext` of the page is set to a new instance of the `SubclassedNativeControlsPageViewModel` class in the code-behind file, with the ViewModel class implementing the `INotifyPropertyChanged` interface.

The page also contains a native picker view for each platform. Each native view displays the collection of fruits by binding its `ItemsSource` property to the `SubclassedNativeControlsPageViewModel.Fruits` collection. This allows the user to pick a fruit, as shown in the following screenshots:



On iOS and Android the native pickers use methods to setup the controls. Therefore, these pickers must be subclassed to expose properties to make them XAML-friendly. On the Universal Windows Platform (UWP), the `ComboBox` is already XAML-friendly, and so doesn't require subclassing.

## iOS

The iOS implementation subclasses the `UIPickerView` view, and exposes properties and an event that can be easily consumed from XAML:

```

public class MyUIPickerView : UIPickerView
{
    public event EventHandler<EventArgs> SelectedItemChanged;

    public MyUIPickerView()
    {
        var model = new PickerModel();
        model.ItemChanged += (sender, e) =>
        {
            if (SelectedItemChanged != null)
            {
                SelectedItemChanged.Invoke(this, e);
            }
        };
        Model = model;
    }

    public IList<string> ItemsSource
    {
        get
        {
            var pickerModel = Model as PickerModel;
            return (pickerModel != null) ? pickerModel.Items : null;
        }
        set
        {
            var model = Model as PickerModel;
            if (model != null)
            {
                model.Items = value;
            }
        }
    }

    public string SelectedItem
    {
        get { return (Model as PickerModel).SelectedItem; }
        set { }
    }
}

```

The `MyUIPickerView` class exposes `ItemsSource` and `SelectedItem` properties, and a `SelectedItemChanged` event. A `UIPickerView` requires an underlying `UIPickerViewModel` data model, which is accessed by the `MyUIPickerView` properties and event. The `UIPickerViewModel` data model is provided by the `PickerModel` class:

```

class PickerModel : UIPickerViewModel
{
    int selectedIndex = 0;
    public event EventHandler<EventArgs> ItemChanged;
    public IList<string> Items { get; set; }

    public string SelectedItem
    {
        get
        {
            return Items != null && selectedIndex >= 0 && selectedIndex < Items.Count ? Items[selectedIndex] : null;
        }
    }

    public override nint GetRowsInComponent(UIPickerView pickerView, nint component)
    {
        return Items != null ? Items.Count : 0;
    }

    public override string GetTitle(UIPickerView pickerView, nint row, nint component)
    {
        return Items != null && Items.Count > row ? Items[(int)row] : null;
    }

    public override nint GetComponentCount(UIPickerView pickerView)
    {
        return 1;
    }

    public override void Selected(UIPickerView pickerView, nint row, nint component)
    {
        selectedIndex = (int)row;
        if (ItemChanged != null)
        {
            ItemChanged.Invoke(this, new EventArgs());
        }
    }
}

```

The `PickerModel` class provides the underlying storage for the `MyUIPickerView` class, via the `Items` property. Whenever the selected item in the `MyUIPickerView` changes, the `Selected` method is executed, which updates the selected index and fires the `ItemChanged` event. This ensures that the `SelectedItem` property will always return the last item picked by the user. In addition, the `PickerModel` class overrides methods that are used to setup the `MyUIPickerView` instance.

## Android

The Android implementation subclasses the `Spinner` view, and exposes properties and an event that can be easily consumed from XAML:

```

class MySpinner : Spinner
{
    ArrayAdapter adapter;
    IList<string> items;

    public IList<string> ItemsSource
    {
        get { return items; }
        set
        {
            if (items != value)
            {
                items = value;
                adapter.Clear();

                foreach (string str in items)
                {
                    adapter.Add(str);
                }
            }
        }
    }

    public string SelectedObject
    {
        get { return (string)GetItemAtPosition(SelectedItemPosition); }
        set
        {
            if (items != null)
            {
                int index = items.IndexOf(value);
                if (index != -1)
                {
                    SetSelection(index);
                }
            }
        }
    }

    public MySpinner(Context context) : base(context)
    {
        ItemSelected += OnBindableSpinnerItemSelected;

        adapter = new ArrayAdapter(context, Android.Resource.Layout.SimpleSpinnerItem);
        adapter.SetDropDownViewResource(Android.Resource.Layout.SimpleSpinnerDropDownItem);
        Adapter = adapter;
    }

    void OnBindableSpinnerItemSelected(object sender, ItemSelectedEventArgs args)
    {
        SelectedObject = (string)GetItemAtPosition(args.Position);
    }
}

```

The `MySpinner` class exposes `ItemsSource` and `SelectedObject` properties, and a `ItemSelected` event. The items displayed by the `MySpinner` class are provided by the `Adapter` associated with the view, and items are populated into the `Adapter` when the `ItemsSource` property is first set. Whenever the selected item in the `MySpinner` class changes, the `OnBindableSpinnerItemSelected` event handler updates the `SelectedObject` property.

## Summary

This article demonstrated how to consume native views from Xamarin.Forms XAML files. Properties and event handlers can be set on native views, and they can interact with Xamarin.Forms views.

## Related Links

- [NativeSwitch \(sample\)](#)
- [Forms2Native \(sample\)](#)
- [NativeViewInsideContentView \(sample\)](#)
- [SubclassedNativeControls \(sample\)](#)
- [Native Forms](#)
- [Passing Arguments in XAML](#)

# Native Views in C#

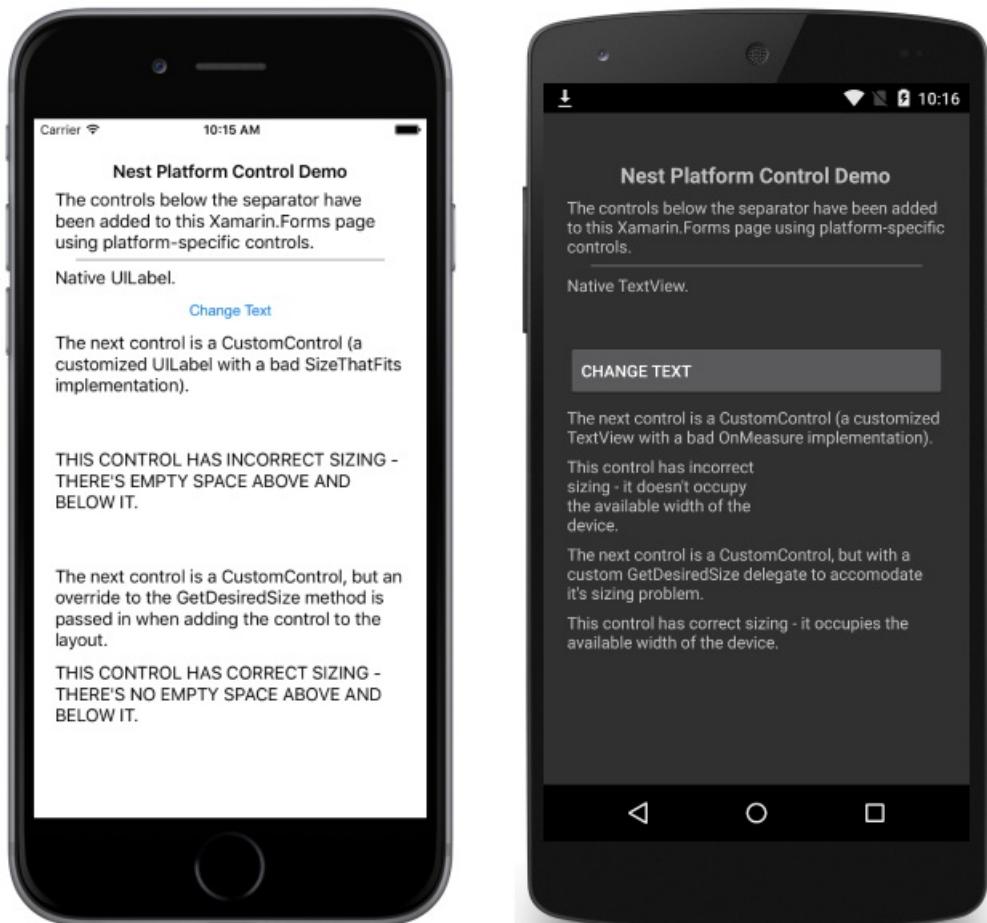
10/9/2018 • 7 minutes to read • [Edit Online](#)

Native views from iOS, Android, and UWP can be directly referenced from Xamarin.Forms pages created using C#. This article demonstrates how to add native views to a Xamarin.Forms layout created using C#, and how to override the layout of custom views to correct their measurement API usage.

## Overview

Any Xamarin.Forms control that allows `Content` to be set, or that has a `Children` collection, can add platform-specific views. For example, an iOS `UILabel` can be directly added to the `ContentView.Content` property, or to the `StackLayout.Children` collection. However, note that this functionality requires the use of `#if` defines in Xamarin.Forms Shared Project solutions, and isn't available from Xamarin.Forms .NET Standard library solutions.

The following screenshots demonstrate platform-specific views having been added to a Xamarin.Forms `StackLayout`:



The ability to add platform-specific views to a Xamarin.Forms layout is enabled by two extension methods on each platform:

- `Add` – adds a platform-specific view to the `Children` collection of a layout.
- `ToView` – takes a platform-specific view and wraps it as a Xamarin.Forms `View` that can be set as the `Content` property of a control.

Using these methods in a Xamarin.Forms shared project requires importing the appropriate platform-specific

Xamarin.Forms namespace:

- **iOS** – Xamarin.Forms.Platform.iOS
- **Android** – Xamarin.Forms.Platform.Android
- **Universal Windows Platform (UWP)** – Xamarin.Forms.Platform.UWP

## Adding Platform-Specific Views on Each Platform

The following sections demonstrate how to add platform-specific views to a Xamarin.Forms layout on each platform.

### iOS

The following code example demonstrates how to add a `UILabel` to a `StackLayout` and a `ContentView`:

```
var uiLabel = new UILabel {
    MinimumFontSize = 14f,
    Lines = 0,
    LineBreakMode = UILineBreakMode.WordWrap,
    Text = originalText,
};
stackLayout.Children.Add (uiLabel);
contentView.Content = uiLabel.ToView();
```

The example assumes that the `stackLayout` and `contentView` instances have previously been created in XAML or C#.

### Android

The following code example demonstrates how to add a `TextView` to a `StackLayout` and a `ContentView`:

```
var textView = new TextView (MainActivity.Instance) { Text = originalText, TextSize = 14 };
stackLayout.Children.Add (textView);
contentView.Content = textView.ToView();
```

The example assumes that the `stackLayout` and `contentView` instances have previously been created in XAML or C#.

### Universal Windows Platform

The following code example demonstrates how to add a `TextBlock` to a `StackLayout` and a `ContentView`:

```
var textBlock = new TextBlock
{
    Text = originalText,
    FontSize = 14,
    FontFamily = new FontFamily("Helvetica Neue"),
    TextWrapping = TextWrapping.Wrap
};
stackLayout.Children.Add(textBlock);
contentView.Content = textBlock.ToView();
```

The example assumes that the `stackLayout` and `contentView` instances have previously been created in XAML or C#.

## Overriding Platform Measurements for Custom Views

Custom views on each platform often only correctly implement measurement for the layout scenario for which they were designed. For example, a custom view may have been designed to only occupy half of the available

width of the device. However, after being shared with other users, the custom view may be required to occupy the full available width of the device. Therefore, it can be necessary to override a custom views measurement implementation when being reused in a Xamarin.Forms layout. For that reason, the `Add` and `ToView` extension methods provide overrides that allow measurement delegates to be specified, which can override the custom view layout when it's added to a Xamarin.Forms layout.

The following sections demonstrate how to override the layout of custom views, to correct their measurement API usage.

## iOS

The following code example shows the `CustomControl` class, which inherits from `UILabel`:

```
public class CustomControl : UILabel
{
    public override string Text {
        get { return base.Text; }
        set { base.Text = value.ToUpper(); }
    }

    public override CGSize SizeThatFits (CGSize size)
    {
        return new CGSize (size.Width, 150);
    }
}
```

An instance of this view is added to a `StackLayout`, as demonstrated in the following code example:

```
var customControl = new CustomControl {
    MinimumFontSize = 14,
    Lines = 0,
    LineBreakMode = UILineBreakMode.WordWrap,
    Text = "This control has incorrect sizing - there's empty space above and below it."
};
stackLayout.Children.Add (customControl);
```

However, because the `CustomControl.SizeThatFits` override always returns a height of 150, the view will be displayed with empty space above and below it, as shown in the following screenshot:

**The next control is a `CustomControl` (a customized `UILabel` with a bad `SizeThatFits` implementation).**

**THIS CONTROL HAS INCORRECT SIZING -  
THERE'S EMPTY SPACE ABOVE AND  
BELOW IT.**

A solution to this problem is to provide a `GetDesiredSizeDelegate` implementation, as demonstrated in the following code example:

```

SizeRequest? FixSize (NativeViewWrapperRenderer renderer, double width, double height)
{
    var uiView = renderer.Control;

    if (uiView == null) {
        return null;
    }

    var constraint = new CGSize (width, height);

    // Let the CustomControl determine its size (which will be wrong)
    var badRect = uiView.SizeThatFits (constraint);

    // Use the width and substitute the height
    return new SizeRequest (new Size (badRect.Width, 70));
}

```

This method uses the width provided by the `CustomControl.SizeThatFits` method, but substitutes the height of 150 for a height of 70. When the `CustomControl` instance is added to the `StackLayout`, the `FixSize` method can be specified as the `GetDesiredSizeDelegate` to fix the bad measurement provided by the `CustomControl` class:

```
stackLayout.Children.Add (customControl, FixSize);
```

This results in the custom view being displayed correctly, without empty space above and below it, as shown in the following screenshot:

**The next control is a `CustomControl`, but an override to the `GetDesiredSize` method is passed in when adding the control to the layout.**

**THIS CONTROL HAS CORRECT SIZING -  
THERE'S NO EMPTY SPACE ABOVE AND  
BELOW IT.**

## Android

The following code example shows the `CustomControl` class, which inherits from `TextView`:

```

public class CustomControl : TextView
{
    public CustomControl (Context context) : base (context)
    {

    }

    protected override void OnMeasure (int widthMeasureSpec, int heightMeasureSpec)
    {
        int width = MeasureSpec.GetSize (widthMeasureSpec);

        // Force the width to half of what's been requested.
        // This is deliberately wrong to demonstrate providing an override to fix it with.
        int widthSpec = MeasureSpec.MakeMeasureSpec (width / 2, MeasureSpec.GetMode (widthMeasureSpec));

        base.OnMeasure (widthSpec, heightMeasureSpec);
    }
}

```

An instance of this view is added to a `StackLayout`, as demonstrated in the following code example:

```

var customControl = new CustomControl (MainActivity.Instance) {
    Text = "This control has incorrect sizing - it doesn't occupy the available width of the device.",
    TextSize = 14
};
stackLayout.Children.Add (customControl);

```

However, because the `CustomControl.OnMeasure` override always returns half of the requested width, the view will be displayed occupying only half the available width of the device, as shown in the following screenshot:

The next control is a `CustomControl` (a customized `TextView` with a bad `OnMeasure` implementation).

This control has incorrect sizing - it doesn't occupy the available width of the device.

A solution to this problem is to provide a `GetDesiredSizeDelegate` implementation, as demonstrated in the following code example:

```

SizeRequest? FixSize (NativeViewWrapperRenderer renderer, int widthConstraint, int heightConstraint)
{
    var nativeView = renderer.Control;

    if ((widthConstraint == 0 && heightConstraint == 0) || nativeView == null) {
        return null;
    }

    int width = Android.Views.View.MeasureSpec.GetSize (widthConstraint);
    int widthSpec = Android.Views.View.MeasureSpec.MakeMeasureSpec (
        width * 2, Android.Views.View.MeasureSpec.GetMode (widthConstraint));
    nativeView.Measure (widthSpec, heightConstraint);
    return new SizeRequest (new Size (nativeView.MeasuredWidth, nativeView.MeasuredHeight));
}

```

This method uses the width provided by the `CustomControl.OnMeasure` method, but multiplies it by two. When the `CustomControl` instance is added to the `StackLayout`, the `FixSize` method can be specified as the `GetDesiredSizeDelegate` to fix the bad measurement provided by the `CustomControl` class:

```
stackLayout.Children.Add (customControl, FixSize);
```

This results in the custom view being displayed correctly, occupying the width of the device, as shown in the following screenshot:

The next control is a `CustomControl`, but with a custom `GetDesiredSize` delegate to accomodate its sizing problem.

This control has correct sizing - it occupies the available width of the device.

## Universal Windows Platform

The following code example shows the `CustomControl` class, which inherits from `Panel`:

```

public class CustomControl : Panel
{
    public static readonly DependencyProperty TextProperty =
        DependencyProperty.Register(
            "Text", typeof(string), typeof(CustomControl), new PropertyMetadata(default(string),
OnTextPropertyChanged));

    public string Text
    {
        get { return (string)GetValue(TextProperty); }
        set { SetValue(TextProperty, value.ToUpper()); }
    }

    readonly TextBlock textBlock;

    public CustomControl()
    {
        textBlock = new TextBlock
        {
            MinHeight = 0,
            MaxHeight = double.PositiveInfinity,
            MinWidth = 0,
            MaxWidth = double.PositiveInfinity,
            FontSize = 14,
            TextWrapping = TextWrapping.Wrap,
            VerticalAlignment = VerticalAlignment.Center
        };

        Children.Add(textBlock);
    }

    static void OnTextPropertyChanged(DependencyObject dependencyObject, DependencyPropertyChangedEventArgs
args)
    {
        ((CustomControl)dependencyObject).textBlock.Text = (string)args.NewValue;
    }

    protected override Size ArrangeOverride(Size finalSize)
    {
        // This is deliberately wrong to demonstrate providing an override to fix it with.
        textBlock.Arrange(new Rect(0, 0, finalSize.Width/2, finalSize.Height));
        return finalSize;
    }

    protected override Size MeasureOverride(Size availableSize)
    {
        textBlock.Measure(availableSize);
        return new Size(textBlock.DesiredSize.Width, textBlock.DesiredSize.Height);
    }
}

```

An instance of this view is added to a `StackLayout`, as demonstrated in the following code example:

```

var brokenControl = new CustomControl {
    Text = "This control has incorrect sizing - it doesn't occupy the available width of the device."
};
stackLayout.Children.Add(brokenControl);

```

However, because the `CustomControl.ArrangeOverride` override always returns half of the requested width, the view will be clipped to half the available width of the device, as shown in the following screenshot:

The next control is a `CustomControl` (a customized `TextBlock` with a bad `ArrangeOverride` implementation).

THIS CONTROL HAS INCORRECT SIZING - IT OCCUPIES THE AVAILABLE WIDTH OF THE DEVICE.

A solution to this problem is to provide an `ArrangeOverrideDelegate` implementation, when adding the view to the `StackLayout`, as demonstrated in the following code example:

```
stackLayout.Children.Add(fixedControl, arrangeOverrideDelegate: (renderer, finalSize) =>
{
    if (finalSize.Width <= 0 || double.IsInfinity(finalSize.Width))
    {
        return null;
    }
    var frameworkElement = renderer.Control;
    frameworkElement.Arrange(new Rect(0, 0, finalSize.Width * 2, finalSize.Height));
    return finalSize;
});
```

This method uses the width provided by the `CustomControl.ArrangeOverride` method, but multiplies it by two. This results in the custom view being displayed correctly, occupying the width of the device, as shown in the following screenshot:

The next control is a `CustomControl`, but an `ArrangeOverride` delegate is passed in when adding the control to the layout.

THIS CONTROL HAS CORRECT SIZING - IT OCCUPIES THE AVAILABLE WIDTH OF THE DEVICE.

## Summary

This article explained how to add native views to a Xamarin.Forms layout created using C#, and how to override the layout of custom views to correct their measurement API usage.

## Related Links

- [NativeEmbedding \(sample\)](#)
- [Native Forms](#)

# Platform-Specifics

11/20/2018 • 2 minutes to read • [Edit Online](#)

*Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects.*

The following platform-specific functionality is provided for Xamarin.Forms views, pages, and layouts:

IOS	ANDROID	WINDOWS
VisualElement.BlurEffect	VisualElement.Elevation	VisualElement.AccessKey, VisualElement.AccessKeyPlacement, VisualElement.AccessKeyHorizontalOffset, and VisualElement.AccessKeyVerticalOffset
VisualElement.IsLegacyColorModeEnabled	VisualElement.IsLegacyColorModeEnabled	VisualElement.IsLegacyColorModeEnabled
VisualElement.IsShadowEnabled		

The following platform-specific functionality is provided for Xamarin.Forms views:

IOS	ANDROID	WINDOWS
Entry.AdjustsFontSizeToFitWidth	Button.UseDefaultPadding and Button.UseDefaultShadow	InputView.DetectReadingOrderFromContent, Label.DetectReadingOrderFromContent
Entry.CursorColor	Entry.ImeOptions	ListView.SelectionMode
ListView.SeparatorStyle	ImageButton.IsShadowEnabled	SearchBar.IsSpellCheckEnabled
Picker.UpdateMode	ListView.IsFastScrollEnabled	WebView.IsJavaScriptAlertEnabled
Slider.UpdateOnTap	WebView.MixedContentMode	

The following platform-specific functionality is provided for Xamarin.Forms pages:

IOS	ANDROID	WINDOWS
NavigationPage.HideSeparatorBar	NavigationPage.BarHeight	MasterDetailPage.CollapsedPaneWidth and MasterDetailPage.CollapseStyle
NavigationPage.IsNavigationBarTransparent	TabbedPage.IsSmoothScrollEnabled	Page.ToolbarPlacement
NavigationPage.StatusBarTextColorMode	TabbedPage.IsSwipePagingEnabled	TabbedPage.HeaderIconsEnabled and TabbedPage.HeaderIconsSize

IOS	ANDROID	WINDOWS
NavigationPage.PreferLargeTitles	TabbedPage.ToolbarPlacement, TabbedPage.BarItemColor, and TabbedPage.BarSelectedItemColor	
Page.ModalPresentationStyle		
Page.PreferStatusBarHidden and Page.PreferredStatusBarUpdateAnimation		
Page.UseSafeArea		

The following platform-specific functionality is provided for Xamarin.Forms layouts:

IOS
ScrollView.ShouldDelayContentTouches

The following platform-specific functionality is provided for the Xamarin.Forms `Application` class:

IOS	ANDROID
Application.HandleControlUpdatesOnMainThread	Application.WindowSoftInputModeAdjust
Application.PanGestureRecognizerShouldRecognizeSimultaneously	Application.SendDisappearingEventOnPause, Application.SendAppearingEventOnResume, and Application.ShouldPreserveKeyboardOnResume

## Consuming platform-specifics

The process for consuming a platform-specific through XAML, or through the fluent code API is as follows:

1. Add a `xmlns` declaration or `using` directive for the `Xamarin.Forms.PlatformConfiguration` namespace.
2. Add a `xmlns` declaration or `using` directive for the namespace that contains the platform-specific functionality:
  - a. On iOS, this is the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace.
  - b. On Android, this is the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace. For Android AppCompat, this is the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` namespace.
  - c. On the Universal Windows Platform, this is the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace.
3. Apply the platform-specific from XAML, or from code with the `on<T>` fluent API. The value of `T` can be the `iOS`, `Android`, or `Windows` types from the `Xamarin.Forms.PlatformConfiguration` namespace.

### NOTE

Note that attempting to consume a platform-specific on a platform where it is unavailable will not result in an error. Instead, the code will execute without the platform-specific being applied.

Platform-specifics consumed through the `on<T>` fluent code API return `IPlatformElementConfiguration` objects. This allows multiple platform-specifics to be invoked on the same object with method cascading.

For more information about platform-specifics, see [Consuming Platform-Specifics](#) and [Creating Platform-](#)

Specifics.

## Related Links

- [Consuming Platform-Specifics](#)
- [Creating Platform-Specifics](#)
- [PlatformSpecifics \(sample\)](#)
- [PlatformConfiguration](#)

# Consuming Platform-Specifics

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Consuming functionality that's only available on a specific platform, without implementing custom renderers or effects.*

## iOS

This article demonstrates how to consume the iOS platform-specifics that are built into Xamarin.Forms.

## Android

This article demonstrates how to consume the Android platform-specifics that are built into Xamarin.Forms.

## Windows

This article demonstrates how to consume the Windows platform-specifics that are built into Xamarin.Forms.

# iOS Platform-Specifics

11/20/2018 • 21 minutes to read • [Edit Online](#)

*Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects. This article demonstrates how to consume the iOS platform-specifics that are built into Xamarin.Forms.*

## VisualElements

On iOS, the following platform-specific functionality is provided for Xamarin.Forms views, pages, and layouts:

- Blur support for any `VisualElement`. For more information, see [Applying Blur](#).
- Disabling legacy color mode on a supported `VisualElement`. For more information, see [Disabling Legacy Color Mode](#).
- Enabling a drop shadow on a `VisualElement`. For more information, see [Enabling a Drop Shadow](#).

### Applying Blur

This platform-specific is used to blur the content layered beneath it, and is consumed in XAML by setting the `VisualElement.BlurEffect` attached property to a value of the `BlurEffectStyle` enumeration:

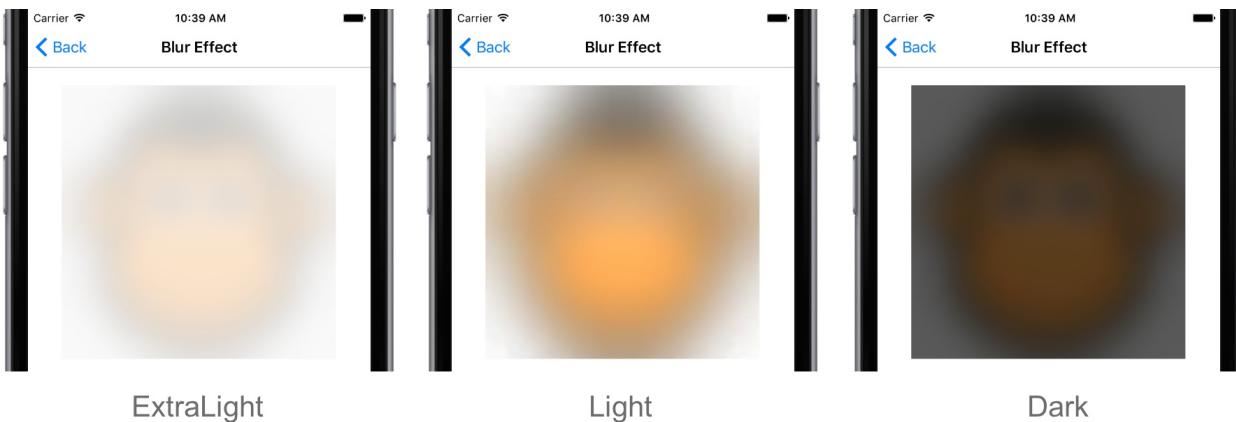
```
<ContentPage ...>
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    ...
    <AbsoluteLayout HorizontalOptions="Center">
        <Image Source="monkeyface.png" />
        <BoxView x:Name="boxView" ios:VisualElement.BlurEffect="ExtraLight" HeightRequest="300"
            WidthRequest="300" />
    </AbsoluteLayout>
    ...
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
boxView.On<iOS>().UseBlurEffect(BlurEffectStyle.ExtraLight);
```

The `BoxView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `VisualElement.UseBlurEffect` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to apply the blur effect, with the `BlurEffectStyle` enumeration providing four values: `None`, `ExtraLight`, `Light`, and `Dark`.

The result is that a specified `BlurEffectStyle` is applied to the `BoxView` instance, which blurs the `Image` layered beneath it:



#### NOTE

When adding a blur effect to a `VisualElement`, touch events will still be received by the `VisualElement`.

### Disabling Legacy Color Mode

Some of the Xamarin.Forms views feature a legacy color mode. In this mode, when the `.IsEnabled` property of the view is set to `false`, the view will override the colors set by the user with the default native colors for the disabled state. For backwards compatibility, this legacy color mode remains the default behavior for supported views.

This platform-specific disables this legacy color mode, so that colors set on a view by the user remain even when the view is disabled. It's consumed in XAML by setting the `VisualElement.IsLegacyColorModeEnabled` attached property to `false`:

```
<ContentPage ...
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core">
    <StackLayout>
        ...
        <Button Text="Button"
            TextColor="Blue"
            BackgroundColor="Bisque"
            ios:VisualElement.IsLegacyColorModeEnabled="False" />
        ...
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOS;
...
_legacyColorModeDisabledButton.On<iOS>().SetIsLegacyColorModeEnabled(false);
```

The `VisualElement.On<iOS>` method specifies that this platform-specific will only run on iOS. The `VisualElement.SetIsLegacyColorModeEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.iOS` namespace, is used to control whether the legacy color mode is disabled. In addition, the `VisualElement.GetIsLegacyColorModeEnabled` method can be used to return whether the legacy color mode is disabled.

The result is that the legacy color mode can be disabled, so that colors set on a view by the user remain even when the view is disabled:

The Button below uses the legacy color mode. When `IsEnabled` is false, it uses the default native colors for the control.

Button

[Toggle IsEnabled \(Currently: False\)](#)

The Button below has the legacy color mode disabled. It will use whatever colors are manually set.

Button

[Toggle IsEnabled \(Currently: False\)](#)

#### NOTE

When setting a `VisualStateGroup` on a view, the legacy color mode is completely ignored. For more information about visual states, see [The Xamarin.Forms Visual State Manager](#).

### Enabling a Drop Shadow

This platform-specific is used to enable a drop shadow on a `VisualElement`. It's consumed in XAML by setting the `VisualElement.IsEnabled` attached property to `true`, along with a number of additional optional attached properties that control the drop shadow:

```
<ContentPage ...>
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core">
        <StackLayout Margin="20">
            <BoxView ...
                ios:VisualElement.IsEnabled="true"
                ios:VisualElement.ShadowColor="Purple"
                ios:VisualElement.ShadowOpacity="0.7"
                ios:VisualElement.ShadowRadius="12">
                <ios:VisualElement.ShadowOffset>
                    <Size>
                        <x:Arguments>
                            <x:Double>10</x:Double>
                            <x:Double>10</x:Double>
                        </x:Arguments>
                    </Size>
                </ios:VisualElement.ShadowOffset>
            </BoxView>
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOS;
...
var boxView = new BoxView { Color = Color.Aqua, WidthRequest = 100, HeightRequest = 100 };
boxView.On<iOS>()
    .SetIsShadowEnabled(true)
    .SetShadowColor(Color.Purple)
    .SetShadowOffset(new Size(10,10))
    .SetShadowOpacity(0.7)
    .SetShadowRadius(12);
```

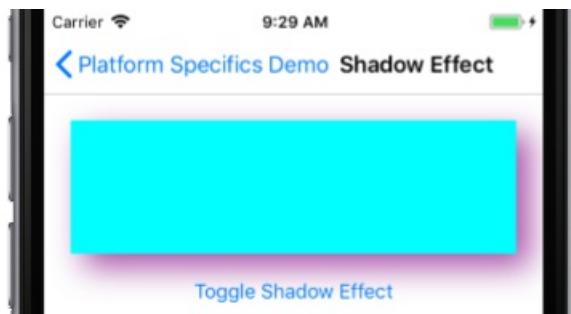
The `VisualElement.On<iOS>` method specifies that this platform-specific will only run on iOS. The `VisualElement.SetIsShadowEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether a drop shadow is enabled on the `VisualElement`. In addition, the following methods can be invoked to control the drop shadow:

- `SetShadowColor` – sets the color of the drop shadow. The default color is `Color.Default`.
- `SetShadowOffset` – sets the offset of the drop shadow. The offset changes the direction the shadow is cast, and is specified as a `Size` value. The `size` structure values are expressed in device-independent units, with the first value being the distance to the left (negative value) or right (positive value), and the second value being the distance above (negative value) or below (positive value). The default value of this property is (0.0, 0.0), which results in the shadow being cast around every side of the `VisualElement`.
- `SetShadowOpacity` – sets the opacity of the drop shadow, with the value being in the range 0.0 (transparent) to 1.0 (opaque). The default opacity value is 0.5.
- `SetShadowRadius` – sets the blur radius used to render the drop shadow. The default radius value is 10.0.

#### NOTE

The state of a drop shadow can be queried by calling the `GetIsShadowEnabled`, `GetShadowColor`, `GetShadowOffset`, `GetShadowOpacity`, and `GetShadowRadius` methods.

The result is that a drop shadow can be enabled on a `VisualElement`:



## Views

On iOS, the following platform-specific functionality is provided for Xamarin.Forms views:

- Ensuring that inputted text fits into an `Entry` by adjusting the font size. For more information, see [Adjusting the Font Size of an Entry](#).
- Setting the cursor color in a `Entry`. For more information, see [Setting the Entry Cursor Color](#).
- Setting the separator style on a `ListView`. For more information, see [Setting the Separator Style on a ListView](#).
- Controlling when item selection occurs in a `Picker`. For more information, see [Controlling Picker Item Selection](#).
- Enabling the `Slider.Value` property to be set by tapping on a position on the `Slider` bar, rather than by having to drag the `Slider` thumb. For more information, see [Enabling a Slider Thumb to Move on Tap](#).

### Adjusting the Font Size of an Entry

This platform-specific is used to scale the font size of an `Entry` to ensure that the inputted text fits in the control. It's consumed in XAML by setting the `Entry.AdjustsFontSizeToFitWidth` attached property to a `boolean` value:

```

<ContentPage ...
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    <StackLayout Margin="20">
        <Entry x:Name="entry"
            Placeholder="Enter text here to see the font size change"
            FontSize="22"
            ios:Entry.AdjustsFontSizeToFitWidth="true" />
        ...
    </StackLayout>
</ContentPage>

```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
entry.On<iOS>().EnableAdjustsFontSizeToFitWidth();

```

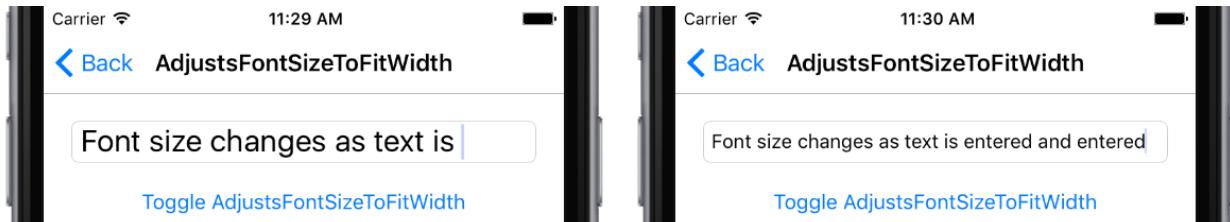
The `Entry.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Entry.EnableAdjustsFontSizeToFitWidth` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to scale the font size of the inputted text to ensure that it fits in the `Entry`. In addition, the `Entry` class in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace also has a `DisableAdjustsFontSizeToFitWidth` method that disables this platform-specific, and a `SetAdjustsFontSizeToFitWidth` method which can be used to toggle font size scaling by calling the `AdjustsFontSizeToFitWidth` method:

```

entry.On<iOS>().SetAdjustsFontSizeToFitWidth(!entry.On<iOS>().AdjustsFontSizeToFitWidth());

```

The result is that the font size of the `Entry` is scaled to ensure that the inputted text fits in the control:



## Setting the Entry Cursor Color

This platform-specific sets the cursor color in an `Entry` to a specified color. It's consumed in XAML by setting the `Entry.CursorColor` bindable property to a `Color`:

```

<ContentPage ...
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <Entry ... ios:Entry.CursorColor="LimeGreen" />
    </StackLayout>
</ContentPage>

```

Alternatively, it can be consumed from C# using the fluent API:

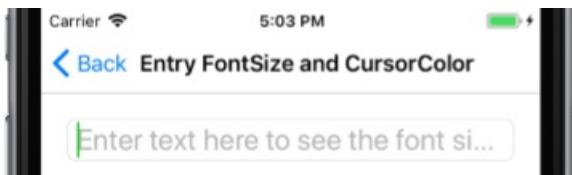
```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
var entry = new Xamarin.Forms.Entry();
entry.On<iOS>().SetCursorColor(Color.LimeGreen);

```

The `Entry.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Entry.SetCursorColor` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, sets the cursor color to a specified `Color`. In addition, the `Entry.GetCursorColor` method can be used to retrieve the current cursor color.

The result is that the cursor color in a `Entry` can be set to a specific `Color`:



### Setting the Separator Style on a ListView

This platform-specific controls whether the separator between cells in a `ListView` uses the full width of the `ListView`. It's consumed in XAML by setting the `ListView.SeparatorStyle` attached property to a value of the `SeparatorStyle` enumeration:

```

<ContentPage ...
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        <ListView ... ios:ListView.SeparatorStyle="FullWidth">
            ...
        </ListView>
    </StackLayout>
</ContentPage>

```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
listView.On<iOS>().SetSeparatorStyle(SeparatorStyle.FullWidth);

```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `ListView.SetSeparatorStyle` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether the separator between cells in the `ListView` uses the full width of the `ListView`, with the `SeparatorStyle` enumeration providing two possible values:

- `Default` – indicates the default iOS separator behavior. This is the default behavior in Xamarin.Forms.
- `FullWidth` – indicates that separators will be drawn from one edge of the `ListView` to the other.

The result is that a specified `SeparatorStyle` value is applied to the `ListView`, which controls the width of the separator between cells:

Abernathy, Cole	55	Auer, Jarret	24
Altenwerth, Serena	32	Auer, Buddy	45
Armstrong, Yoshiko	56	Anderson, Carlie	38
Ankunding, Wilson	49	Adams, Penelope	63
Ankunding, Ray	42	Abbott, King	26
Anderson, Wilburn	63	Auer, Jerrold	31

## SeparatorStyle.Default   SeparatorStyle.FullWidth

### NOTE

Once the separator style has been set to `FullWidth`, it cannot be changed back to `Default` at runtime.

### Controlling Picker Item Selection

This platform-specific controls when item selection occurs in a `Picker`, allowing the user to specify that item selection occurs when browsing items in the control, or only once the **Done** button is pressed. It's consumed in XAML by setting the `Picker.UpdateMode` attached property to a value of the `updateMode` enumeration:

```
<ContentPage ...
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        <Picker ... Title="Select a monkey" ios:Picker.UpdateMode="WhenFinished">
            ...
        </Picker>
        ...
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOS;
...
picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

The `Picker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Picker.SetUpdateMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOS` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `Picker`. This is the default behavior in Xamarin.Forms.
- `WhenFinished` – item selection only occurs once the user has pressed the **Done** button in the `picker`.

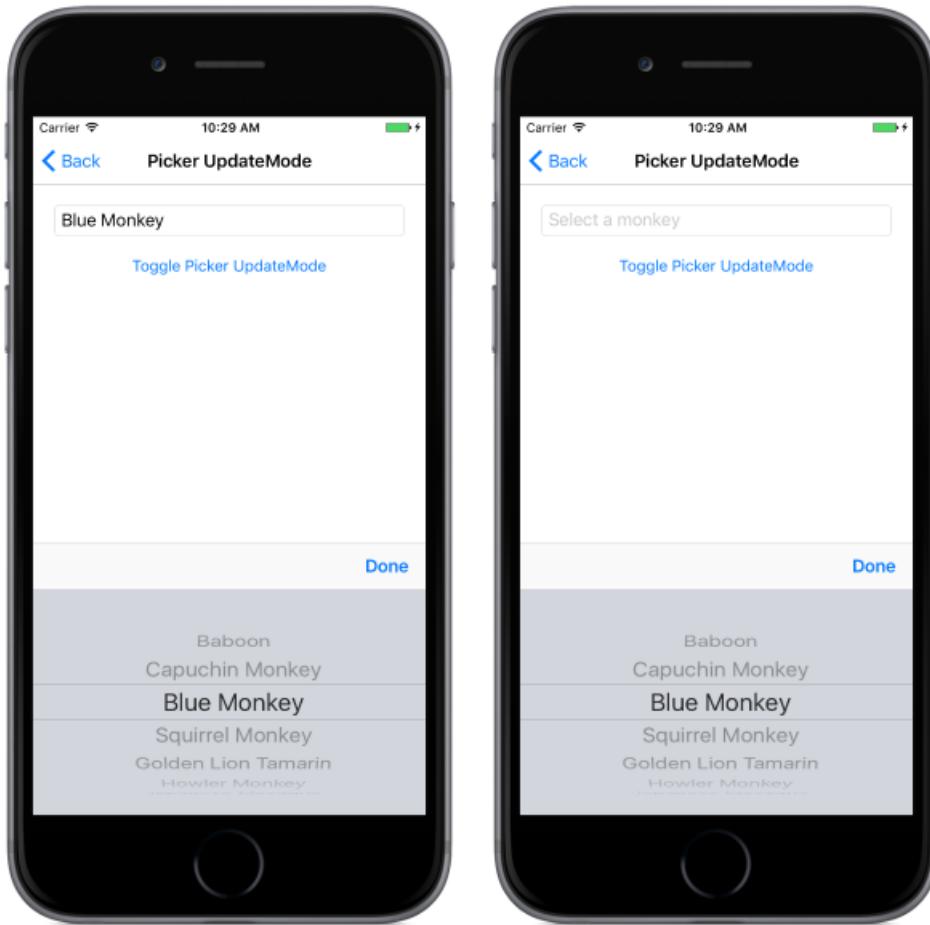
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `UpdateMode`:

```

switch (picker.On<iOS>().UpdateMode())
{
    case UpdateMode.Immediately:
        picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
        break;
    case UpdateMode.WhenFinished:
        picker.On<iOS>().SetUpdateMode(UpdateMode.Immediately);
        break;
}

```

The result is that a specified `UpdateMode` is applied to the `Picker`, which controls when item selection occurs:



`EditMode.Immediately`

`EditMode.WhenFinished`

### Enabling a Slider Thumb to Move on Tap

This platform-specific enables the `Slider.Value` property to be set by tapping on a position on the `Slider` bar, rather than by having to drag the `Slider` thumb. It's consumed in XAML by setting the `Slider.UpdateOnTap` bindable property to `true`:

```

<ContentPage ...
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core">
    <StackLayout ...
        <Slider ... ios:Slider.UpdateOnTap="true" />
        ...
    </StackLayout>
</ContentPage>

```

Alternatively, it can be consumed from C# using the fluent API:

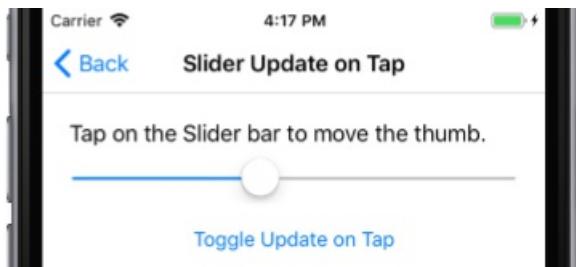
```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
var slider = new Xamarin.Forms.Slider();
slider.On<iOS>().SetUpdateOnTap(true);

```

The `Slider.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Slider.SetUpdateOnTap` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether a tap on the `Slider` bar will set the `Slider.Value` property. In addition, the `Slider.GetUpdateOnTap` method can be used to return whether a tap on the `Slider` bar will set the `Slider.Value` property.

The result is that a tap on the `Slider` bar can move the `Slider` thumb and set the `Slider.Value` property:



## Pages

On iOS, the following platform-specific functionality is provided for Xamarin.Forms pages:

- Hiding the navigation bar separator on a `NavigationPage`. For more information, see [Hiding the Navigation Bar Separator on a NavigationPage](#).
- Controlling whether the navigation bar is translucent. For more information, see [Making the Navigation Bar Translucent](#).
- Controlling whether the status bar text color on a `NavigationPage` is adjusted to match the luminosity of the navigation bar. For more information, see [Adjusting the Status Bar Text Color Mode](#).
- Controlling whether the page title is displayed as a large title in the page navigation bar. For more information, see [Displaying Large Titles](#).
- Setting the status bar visibility on a `Page`. For more information, see [Setting the Status Bar Visibility on a Page](#).
- Ensuring that page content is positioned on an area of the screen that is safe for all iOS devices. For more information, see [Enabling the Safe Area Layout Guide](#).
- Setting the presentation style of modal pages on an iPad. For more information, see [Setting the Modal Page Presentation Style on an iPad](#).

### Hiding the Navigation Bar Separator on a NavigationPage

This platform-specific hides the separator line and shadow that is at the bottom of the navigation bar on a `NavigationPage`. It's consumed in XAML by setting the `NavigationPage.HideNavigationBarSeparator` bindable property to `false`:

```

<NavigationPage ...
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:NavigationPage.HideNavigationBarSeparator="true">

</NavigationPage>

```

Alternatively, it can be consumed from C# using the fluent API:

```

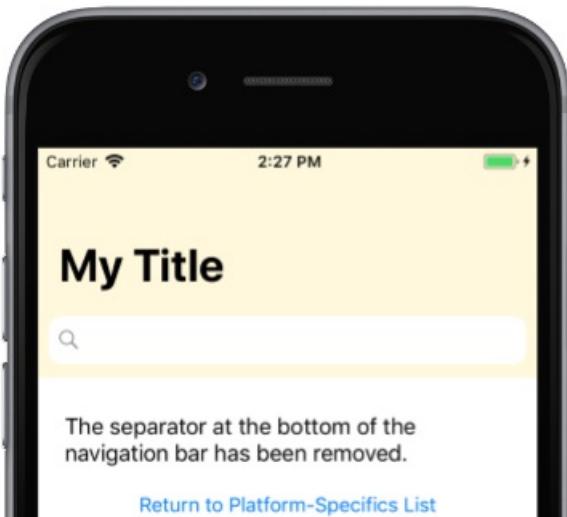
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;

public class iOSTitleViewNavigationPageCS : Xamarin.Forms.NavigationPage
{
    public iOSTitleViewNavigationPageCS()
    {
        On<iOS>().SetHideNavigationBarSeparator(true);
    }
}

```

The `NavigationPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The `NavigationPage.SetHideNavigationBarSeparator` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether the navigation bar separator is hidden. In addition, the `NavigationPage.HideNavigationBarSeparator` method can be used to return whether the navigation bar separator is hidden.

The result is that the navigation bar separator on a `NavigationPage` can be hidden:



## Making the Navigation Bar Translucent

This platform-specific is used to change the transparency of the navigation bar, and is consumed in XAML by setting the `NavigationPage.IsNavigationBarTranslucent` attached property to a `boolean` value:

```

<NavigationView ...
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    BackgroundColor="Blue"
    ios:NavigationPage.IsNavigationBarTranslucent="true">
    ...
</NavigationView>

```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
(App.Current.MainPage as Xamarin.Forms.NavigationPage).BackgroundColor = Color.Blue;
(App.Current.MainPage as Xamarin.Forms.NavigationPage).On<iOS>().EnableTranslucentNavigationBar();

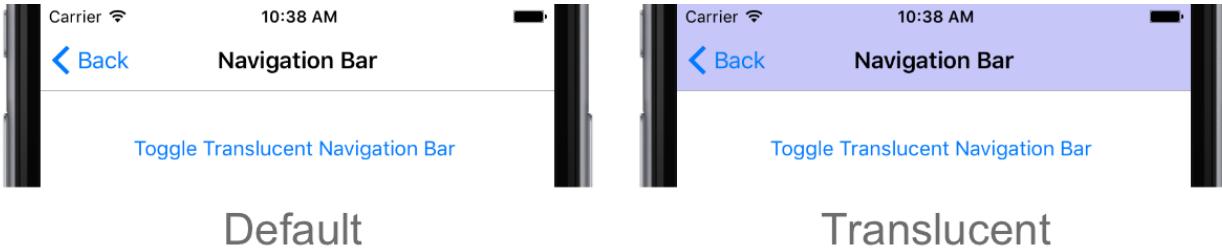
```

The `NavigationPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The

`NavigationPage.EnableTranslucentNavigationBar` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to make the navigation bar translucent. In addition, the `NavigationPage` class in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace also has a `DisableTranslucentNavigationBar` method that restores the navigation bar to its default state, and a `SetIsNavigationBarTranslucent` method which can be used to toggle the navigation bar transparency by calling the `IsNavigationBarTranslucent` method:

```
(App.Current.MainPage as Xamarin.Forms.NavigationPage)
    .On<iOS>()
    .SetIsNavigationBarTranslucent(!((App.Current.MainPage as Xamarin.Forms.NavigationPage).On<iOS>
() .IsNavigationBarTranslucent()));
```

The result is that the transparency of the navigation bar can be changed:



### Adjusting the Status Bar Text Color Mode

This platform-specific controls whether the status bar text color on a `NavigationPage` is adjusted to match the luminosity of the navigation bar. It's consumed in XAML by setting the `NavigationPage.StatusBarTextColorMode` attached property to a value of the `StatusBarTextColorMode` enumeration:

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    x:Class="PlatformSpecifics.iOSStatusBarTextColorModePage">
    <MasterDetailPage.Master>
        <ContentPage Title="Master Page Title" />
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <NavigationPage BarBackgroundColor="Blue" BarTextColor="White"
            ios:NavigationPage.StatusBarTextColorMode="MatchNavigationBarTextLuminosity">
            <x:Arguments>
                <ContentPage>
                    <Label Text="Slide the master page to see the status bar text color mode change." />
                </ContentPage>
            </x:Arguments>
        </NavigationPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
IsPresentedChanged += (sender, e) =>
{
    var mdp = sender as MasterDetailPage;
    if (mdp.IsPresented)
        ((Xamarin.Forms.NavigationPage)mdp.Detail)
            .On<iOS>()
            .SetStatusBarTextColorMode(StatusBarTextColorMode.DoNotAdjust);
    else
        ((Xamarin.Forms.NavigationPage)mdp.Detail)
            .On<iOS>()
            .SetStatusBarTextColorMode(StatusBarTextColorMode.MatchNavigationBarTextLuminosity);
};

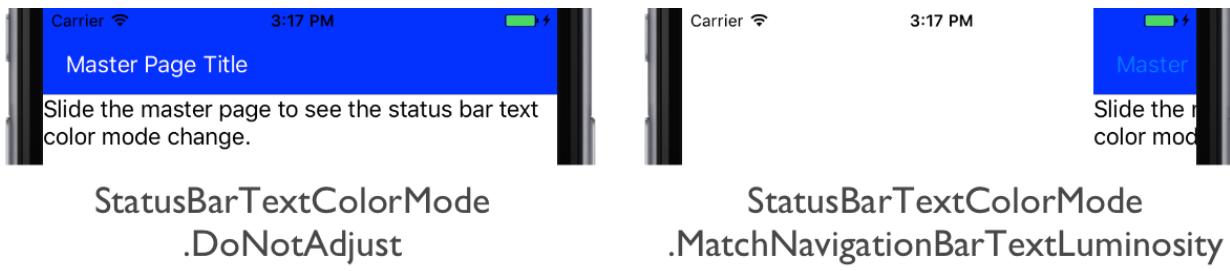
```

The `NavigationPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The `NavigationPage.SetStatusBarTextColorMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, controls whether the status bar text color on the `NavigationPage` is adjusted to match the luminosity of the navigation bar, with the `StatusbarTextColorMode` enumeration providing two possible values:

- `DoNotAdjust` – indicates that the status bar text color should not be adjusted.
- `MatchNavigationBarTextLuminosity` – indicates that the status bar text color should match the luminosity of the navigation bar.

In addition, the `GetStatusBarTextColorMode` method can be used to retrieve the current value of the `StatusbarTextColorMode` enumeration that's applied to the `NavigationPage`.

The result is that the status bar text color on a `NavigationPage` can be adjusted to match the luminosity of the navigation bar. In this example, the status bar text color changes as the user switches between the `Master` and `Detail` pages of a `MasterDetailPage`:



## Displaying Large Titles

This platform-specific is used to display the page title as a large title on the navigation bar, for devices that use iOS 11 or greater. A large title is left aligned and uses a larger font, and transitions to a standard title as the user begins scrolling content, so that the screen real estate is used efficiently. However, in landscape orientation, the title will return to the center of the navigation bar to optimize content layout. It's consumed in XAML by setting the `NavigationPage.PrefersLargeTitles` attached property to a `boolean` value:

```

<NavigationPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ...
    ios:NavigationPage.PrefersLargeTitles="true">
    ...
</NavigationPage>

```

Alternatively it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
var navigationPage = new Xamarin.Forms.NavigationPage(new iOSLargeTitlePageCS());
navigationPage.On<iOS>().SetPrefersLargeTitles(true);
```

The `NavigationPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The `NavigationPage.SetPrefersLargeTitle` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, controls whether large titles are enabled.

Provided that large titles are enabled on the `NavigationPage`, all pages in the navigation stack will display large titles. This behavior can be overridden on pages by setting the `Page.LargeTitleDisplay` attached property to a value of the `LargeTitleDisplayStyle` enumeration:

```
<ContentPage ...
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    Title="Large Title"
    ios:Page.LargeTitleDisplay="Never">
    ...
</ContentPage>
```

Alternatively, the page behavior can be overridden from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
public class iOSLargeTitlePageCS : ContentPage
{
    public iOSLargeTitlePageCS(ICommand restore)
    {
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayStyle.Never);
        ...
    }
    ...
}
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetLargeTitleDisplay` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, controls the large title behavior on the `Page`, with the `LargeTitleDisplayStyle` enumeration providing three possible values:

- `Always` – force the navigation bar and font size to use the large format.
- `Automatic` – use the same style (large or small) as the previous item in the navigation stack.
- `Never` – force the use of the regular, small format navigation bar.

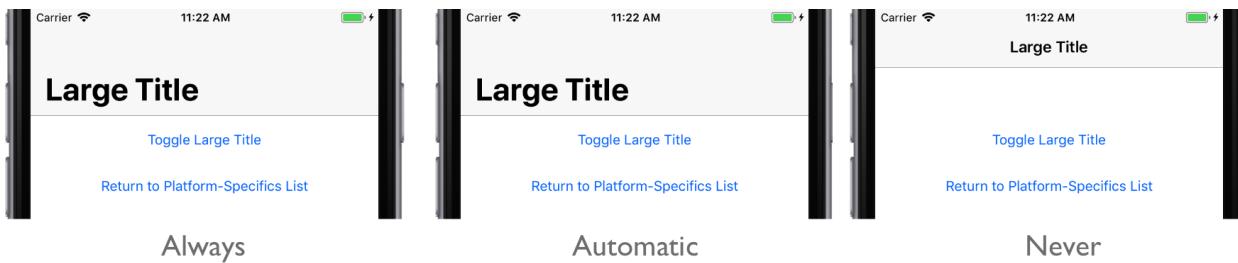
In addition, the `SetLargeTitleDisplay` method can be used to toggle the enumeration values by calling the `LargeTitleDisplay` method, which returns the current `LargeTitleDisplayStyle`:

```

switch (On<iOS>().LargeTitleDisplay())
{
    case LargeTitleDisplayMode.Always:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Automatic);
        break;
    case LargeTitleDisplayMode.Automatic:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Never);
        break;
    case LargeTitleDisplayMode.Never:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Always);
        break;
}

```

The result is that a specified `LargeTitleDisplayMode` is applied to the `Page`, which controls the large title behavior:



## Setting the Status Bar Visibility on a Page

This platform-specific is used to set the visibility of the status bar on a `Page`, and it includes the ability to control how the status bar enters or leaves the `Page`. It's consumed in XAML by setting the `Page.PrefersStatusBarHidden` attached property to a value of the `StatusBarHiddenMode` enumeration, and optionally the `Page.PreferredStatusBarUpdateAnimation` attached property to a value of the `UIStatusBarAnimation` enumeration:

```

<ContentPage ...
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.PrefersStatusBarHidden="True"
    ios:Page.PreferredStatusBarUpdateAnimation="Fade">
    ...
</ContentPage>

```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
On<iOS>().SetPreferedStatusBarHidden(StatusBarHiddenMode.True)
    .SetPreferredStatusBarUpdateAnimation(UIStatusBarAnimation.Fade);

```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetPreferedStatusBarHidden` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to set the visibility of the status bar on a `Page` by specifying one of the `StatusBarHiddenMode` enumeration values: `Default`, `True`, or `False`. The `StatusBarHiddenMode.True` and `StatusBarHiddenMode.False` values set the status bar visibility regardless of device orientation, and the `StatusBarHiddenMode.Default` value hides the status bar in a vertically compact environment.

The result is that the visibility of the status bar on a `Page` can be set:



StatusBarHiddenMode.False

StatusBarHiddenMode.True

#### NOTE

On a `TabbedPage`, the specified `StatusBarHiddenMode` enumeration value will also update the status bar on all child pages. On all other `Page`-derived types, the specified `StatusBarHiddenMode` enumeration value will only update the status bar on the current page.

The `Page.SetPreferredStatusBarUpdateAnimation` method is used to set how the status bar enters or leaves the `Page` by specifying one of the `UIStatusBarAnimation` enumeration values: `None`, `Fade`, or `Slide`. If the `Fade` or `Slide` enumeration value is specified, a 0.25 second animation executes as the status bar enters or leaves the `Page`.

#### Enabling the Safe Area Layout Guide

This platform-specific is used to ensure that page content is positioned on an area of the screen that is safe for all devices that use iOS 11 and greater. Specifically, it will help to make sure that content isn't clipped by rounded device corners, the home indicator, or the sensor housing on an iPhone X. It's consumed in XAML by setting the `Page.UseSafeArea` attached property to a `boolean` value:

```
<ContentPage ...>
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    Title="Safe Area"
    ios:Page.UseSafeArea="true">
    <StackLayout>
        ...
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
On<iOS>().SetUseSafeArea(true);
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetUseSafeArea` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, controls whether the safe area layout guide is enabled.

The result is that page content can be positioned on an area of the screen that is safe for all iPhones:



Enabled



Disabled

#### NOTE

The safe area defined by Apple is used in Xamarin.Forms to set the `Page.Padding` property, and will override any previous values of this property that have been set.

The safe area can be customized by retrieving its `Thickness` value with the `Page.SafeAreaInsets` method from the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace. It can then be modified as required and re-assigned to the `Padding` property in the page constructor or `OnAppearing` override:

```
protected override void OnAppearing()
{
    base.OnAppearing();

    var safeInsets = On<iOS>().SafeAreaInsets();
    safeInsets.Left = 20;
    Padding = safeInsets;
}
```

#### Setting the Modal Page Presentation Style on an iPad

This platform-specific is used to set the presentation style of a modal page on an iPad. It's consumed in XAML by setting the `Page.ModalPresentationStyle` bindable property to a `UIModalPresentationStyle` enumeration value:

```
<ContentPage ...
    xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.ModalPresentationStyle="FormSheet">
...
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
public class iOSModalFormSheetPageCS : ContentPage
{
    public iOSModalFormSheetPageCS()
    {
        On<iOS>().SetModalPresentationStyle(UIModalPresentationStyle.FormSheet);
        ...
    }
}

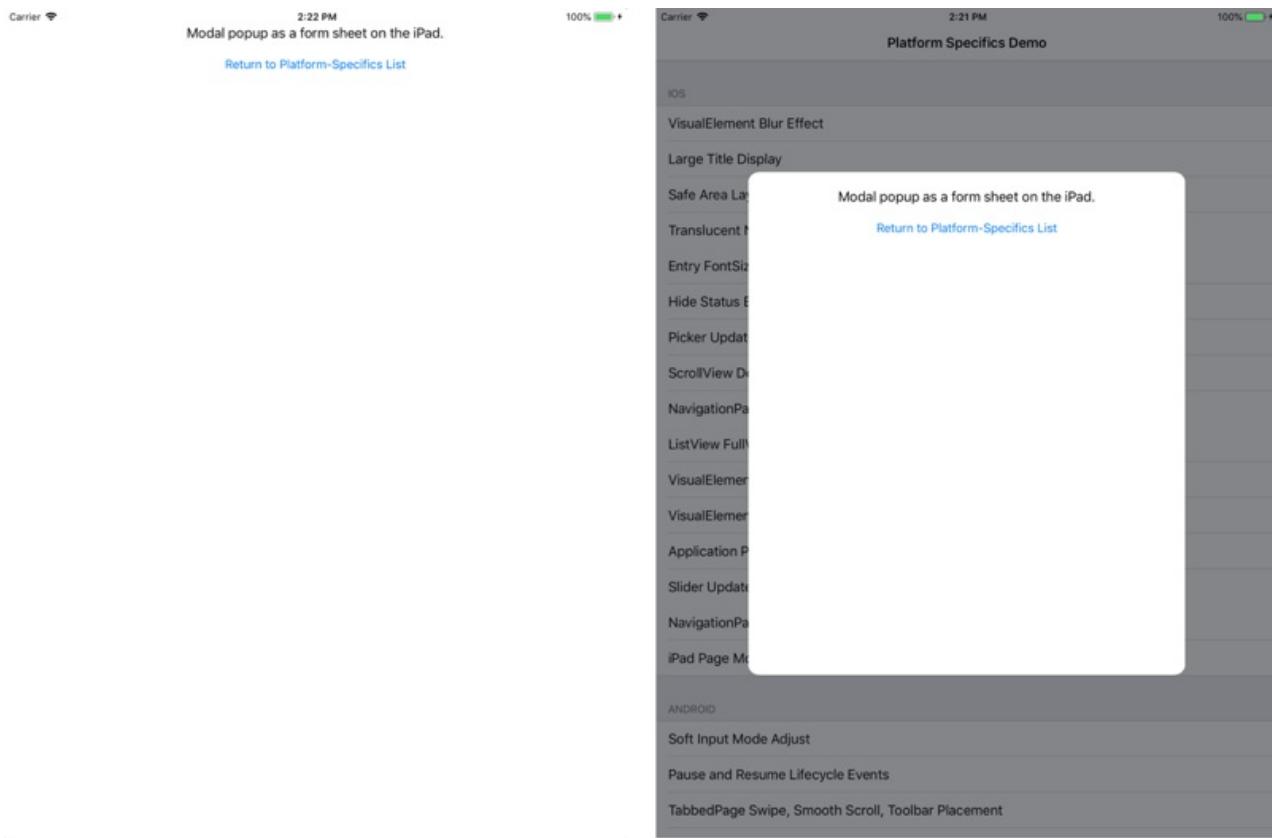
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetModalPresentationStyle` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to set the modal presentation style on a `Page` by specifying one of the following `UIModalPresentationStyle` enumeration values:

- `FullScreen`, which sets the modal presentation style to encompass the whole screen. By default, modal pages are displayed using this presentation style.
- `FormSheet`, which sets the modal presentation style to be centered on and smaller than the screen.

In addition, the `GetModalPresentationStyle` method can be used to retrieve the current value of the `UIModalPresentationStyle` enumeration that's applied to the `Page`.

The result is that the modal presentation style on a `Page` can be set:



FullScreen

FormSheet

#### NOTE

Pages that use this platform-specific to set the modal presentation style must use modal navigation. For more information, see [Xamarin.Forms Modal Pages](#).

# Layouts

On iOS, the following platform-specific functionality is provided for Xamarin.Forms layouts:

- Controlling whether a `ScrollView` handles a touch gesture or passes it to its content. For more information, see [Delaying Content Touches in a ScrollView](#).

## Delaying Content Touches in a ScrollView

An implicit timer is triggered when a touch gesture begins in a `ScrollView` on iOS and the `ScrollView` decides, based on the user action within the timer span, whether it should handle the gesture or pass it to its content. By default, the iOS `ScrollView` delays content touches, but this can cause problems in some circumstances with the `ScrollView` content not winning the gesture when it should. Therefore, this platform-specific controls whether a `ScrollView` handles a touch gesture or passes it to its content. It's consumed in XAML by setting the `ScrollView.ShouldDelayContentTouches` attached property to a `boolean` value:

```
<MasterDetailPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <MasterDetailPage.Master>  
        <ContentPage Title="Menu" BackgroundColor="Blue" />  
    </MasterDetailPage.Master>  
    <MasterDetailPage.Detail>  
        <ContentPage>  
            <ScrollView x:Name="scrollView" ios:ScrollView.ShouldDelayContentTouches="false">  
                <StackLayout Margin="0,20">  
                    <Slider />  
                    <Button Text="Toggle ScrollView DelayContentTouches" Clicked="OnButtonClicked" />  
                </StackLayout>  
            </ScrollView>  
        </ContentPage>  
    </MasterDetailPage.Detail>  
</MasterDetailPage>
```

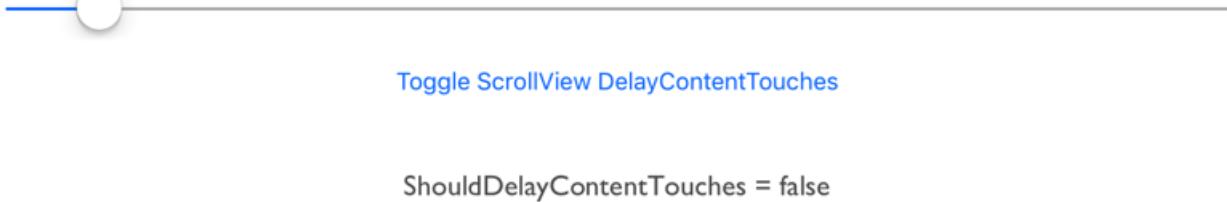
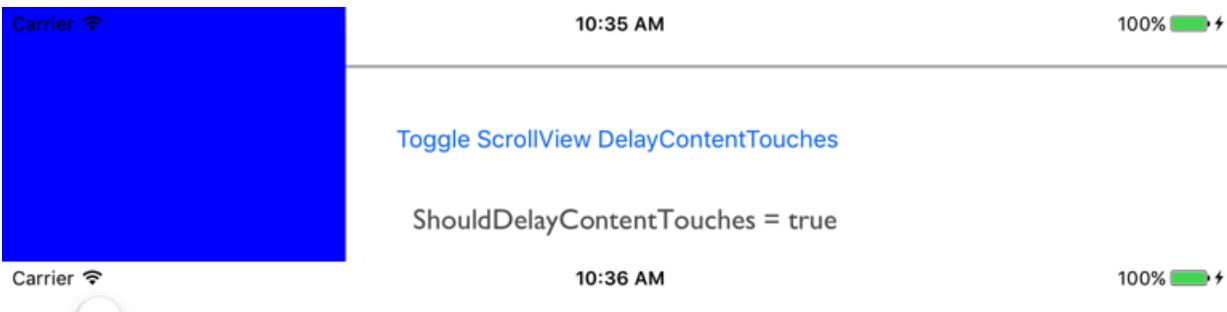
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
scrollView.On<iOS>().SetShouldDelayContentTouches(false);
```

The `ScrollView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `ScrollView.SetShouldDelayContentTouches` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether a `ScrollView` handles a touch gesture or passes it to its content. In addition, the `SetShouldDelayContentTouches` method can be used to toggle delaying content touches by calling the `ShouldDelayContentTouches` method to return whether content touches are delayed:

```
scrollView.On<iOS>().SetShouldDelayContentTouches(!scrollView.On<iOS>().ShouldDelayContentTouches());
```

The result is that a `ScrollView` can disable delaying receiving content touches, so that in this scenario the `Slider` receives the gesture rather than the `Detail` page of the `MasterDetailPage`:



## Application

On iOS, the following platform-specific functionality is provided for the `Xamarin.Forms Application` class:

- Enabling control layout and rendering updates to be performed on the main thread. For more information, see [Handling Control Updates on the Main Thread](#).
- Enabling a `PanGestureRecognizer` in a scrolling view to capture and share the pan gesture with the scrolling view. For more information, see [Enabling Simultaneous Pan Gesture Recognition](#).

### Handling Control Updates on the Main Thread

This platform-specific enables control layout and rendering updates to be performed on the main thread, instead of being performed on a background thread. It should be rarely needed, but in some cases may prevent crashes. Its consumed in XAML by setting the `Application.HandleControlUpdatesOnMainThread` bindable property to `true`:

```
<Application ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:Application.HandleControlUpdatesOnMainThread="true">  
    ...  
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
Xamarin.Forms.Application.Current.On<iOS>().SetHandleControlUpdatesOnMainThread(true);
```

The `Application.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Application.SetHandleControlUpdatesOnMainThread` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether control layout and rendering updates are performed on the main thread, instead of being performed on a background thread. In addition, the `Application.GetHandleControlUpdatesOnMainThread` method can be used to return whether control layout and rendering updates are being performed on the main thread.

### Enabling Simultaneous Pan Gesture Recognition

When a `PanGestureRecognizer` is attached to a view inside a scrolling view, all of the pan gestures are captured by the `PanGestureRecognizer` and aren't passed to the scrolling view. Therefore, the scrolling view will no longer scroll.

This platform-specific enables a `PanGestureRecognizer` in a scrolling view to capture and share the pan gesture with the scrolling view. It's consumed in XAML by setting the

`Application.PanGestureRecognizerShouldRecognizeSimultaneously` attached property to `true`:

```
<Application ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:Application.PanGestureRecognizerShouldRecognizeSimultaneously="true">  
    ...  
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
Xamarin.Forms.Application.Current.On<iOS>().SetPanGestureRecognizerShouldRecognizeSimultaneously(true);
```

The `Application.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Application.SetPanGestureRecognizerShouldRecognizeSimultaneously` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether a pan gesture recognizer in a scrolling view will capture the pan gesture, or capture and share the pan gesture with the scrolling view. In addition, the `Application.GetPanGestureRecognizerShouldRecognizeSimultaneously` method can be used to return whether the pan gesture is shared with the scrolling view that contains the `PanGestureRecognizer`.

Therefore, with this platform-specific enabled, when a `ListView` contains a `PanGestureRecognizer`, both the `ListView` and the `PanGestureRecognizer` will receive the pan gesture and process it. However, with this platform-specific disabled, when a `ListView` contains a `PanGestureRecognizer`, the `PanGestureRecognizer` will capture the pan gesture and process it, and the `ListView` won't receive the pan gesture.

## Summary

This article demonstrated how to consume the iOS platform-specifics that are built into Xamarin.Forms. Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects.

## Related Links

- [Creating Platform-Specifics](#)
- [PlatformSpecifics \(sample\)](#)
- [iOS Specific](#)

# Android Platform-Specifics

11/20/2018 • 16 minutes to read • [Edit Online](#)

*Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects. This article demonstrates how to consume the Android platform-specifics that are built into Xamarin.Forms.*

## VisualElements

On Android, the following platform-specific functionality is provided for Xamarin.Forms views, pages, and layouts:

- Controlling the Z-order of visual elements to determine drawing order. For more information, see [Controlling the Elevation of Visual Elements](#).
- Disabling legacy color mode on a supported [VisualElement](#). For more information, see [Disabling Legacy Color Mode](#).

### Controlling the Elevation of Visual Elements

This platform-specific is used to control the elevation, or Z-order, of visual elements on applications that target API 21 or greater. The elevation of a visual element determines its drawing order, with visual elements with higher Z values occluding visual elements with lower Z values. It's consumed in XAML by setting the

`VisualElement.Elevation` attached property to a `boolean` value:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
    Title="Elevation">
    <StackLayout>
        <Grid>
            <Button Text="Button Beneath BoxView" />
            <BoxView Color="Red" Opacity="0.2" HeightRequest="50" />
        </Grid>
        <Grid Margin="0,20,0,0">
            <Button Text="Button Above BoxView - Click Me" android:VisualElement.Elevation="10"/>
            <BoxView Color="Red" Opacity="0.2" HeightRequest="50" />
        </Grid>
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```

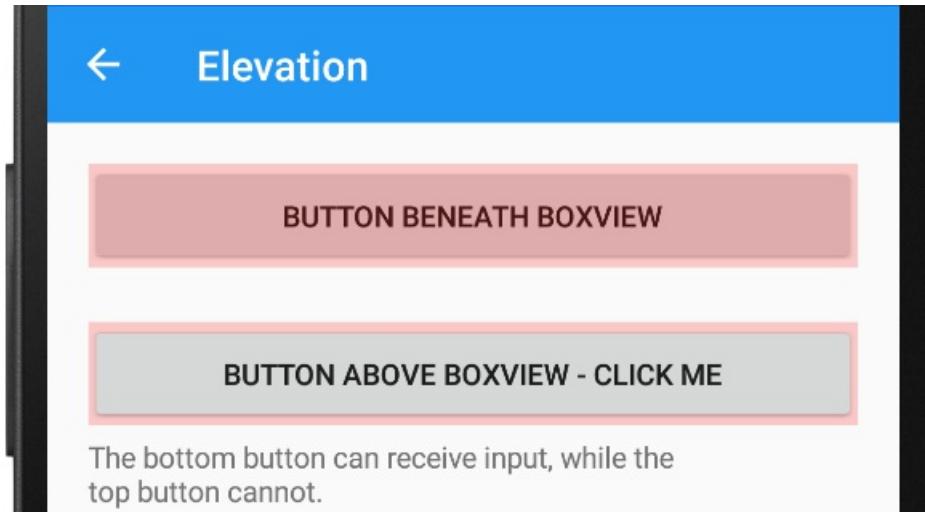
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
public class AndroidElevationPageCS : ContentPage
{
    public AndroidElevationPageCS()
    {
        ...
        var aboveButton = new Button { Text = "Button Above BoxView - Click Me" };
        aboveButton.On<Android>().SetElevation(10);

        Content = new StackLayout
        {
            Children =
            {
                new Grid
                {
                    Children =
                    {
                        new Button { Text = "Button Beneath BoxView" },
                        new BoxView { Color = Color.Red, Opacity = 0.2, HeightRequest = 50 }
                    }
                },
                new Grid
                {
                    Margin = new Thickness(0,20,0,0),
                    Children =
                    {
                        aboveButton,
                        new BoxView { Color = Color.Red, Opacity = 0.2, HeightRequest = 50 }
                    }
                }
            }
        };
    }
}

```

The `Button.On<Android>` method specifies that this platform-specific will only run on Android. The `VisualElement.SetElevation` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to set the elevation of the visual element to a nullable `float`. In addition, the `VisualElement.GetElevation` method can be used to retrieve the elevation value of a visual element.

The result is that the elevation of visual elements can be controlled so that visual elements with higher Z values occlude visual elements with lower Z values. Therefore, in this example the second `Button` is rendered above the `BoxView` because it has a higher elevation value:



## Disabling Legacy Color Mode

Some of the Xamarin.Forms views feature a legacy color mode. In this mode, when the `IsEnabled` property of the view is set to `false`, the view will override the colors set by the user with the default native colors for the disabled state. For backwards compatibility, this legacy color mode remains the default behavior for supported views.

This platform-specific disables this legacy color mode, so that colors set on a view by the user remain even when the view is disabled. It's consumed in XAML by setting the `VisualElement.IsEnabled` attached property to `false`:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            ...
            <Button Text="Button"
                TextColor="Blue"
                BackgroundColor="Bisque"
                android:VisualElement.IsEnabled="False" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
_legacyColorModeDisabledButton.On<Android>().SetIsLegacyColorModeEnabled(false);
```

The `VisualElement.On<Android>` method specifies that this platform-specific will only run on Android. The `VisualElement.SetIsLegacyColorModeEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to control whether the legacy color mode is disabled. In addition, the `VisualElement.GetIsLegacyColorModeEnabled` method can be used to return whether the legacy color mode is disabled.

The result is that the legacy color mode can be disabled, so that colors set on a view by the user remain even when the view is disabled:



### NOTE

When setting a `VisualStateGroup` on a view, the legacy color mode is completely ignored. For more information about visual states, see [The Xamarin.Forms Visual State Manager](#).

# Views

On Android, the following platform-specific functionality is provided for Xamarin.Forms views:

- Using the default padding and shadow values of Android buttons. For more information, see [Using Android Buttons](#).
- Setting the input method editor options for the soft keyboard for an `Entry`. For more information, see [Setting Entry Input Method Editor Options](#).
- Enabling a drop shadow on a `ImageButton`. For more information, see [Enabling a Drop Shadow on a ImageButton](#).
- Enabling fast scrolling in a `ListView`. For more information, see [Enabling Fast Scrolling in a ListView](#).
- Controlling whether a `WebView` can display mixed content. For more information, see [Enabling Mixed Content in a WebView](#).

## Using Android Buttons

This platform-specific controls whether Xamarin.Forms buttons use the default padding and shadow values of Android buttons. It's consumed in XAML by setting the `Button.UseDefaultPadding` and `Button.UseDefaultShadow` attached properties to `boolean` values:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            ...
            <Button ...>
                android:Button.UseDefaultPadding="true"
                android:Button.UseDefaultShadow="true" />
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
button.On<Android>().SetUseDefaultPadding(true).SetUseDefaultShadow(true);
```

The `Button.On<Android>` method specifies that this platform-specific will only run on Android. The `Button.SetUseDefaultPadding` and `Button.SetUseDefaultShadow` methods, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, are used to control whether Xamarin.Forms buttons use the default padding and shadow values of Android buttons. In addition, the `Button.UseDefaultPadding` and `Button.UseDefaultShadow` methods can be used to return whether a button uses the default padding value and default shadow value, respectively.

The result is that Xamarin.Forms buttons can use the default padding and shadow values of Android buttons:



Note that in the screenshot above each `Button` has identical definitions, except that the right-hand `Button` uses the default padding and shadow values of Android buttons.

## Setting Entry Input Method Editor Options

This platform-specific sets the input method editor (IME) options for the soft keyboard for an `Entry`. This includes setting the user action button in the bottom corner of the soft keyboard, and the interactions with the `Entry`. It's consumed in XAML by setting the `Entry.ImeOptions` attached property to a value of the `ImeFlags` enumeration:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout ...>
            <Entry ... android:Entry.ImeOptions="Send" />
            ...
        </StackLayout>
    </ContentPage>
```

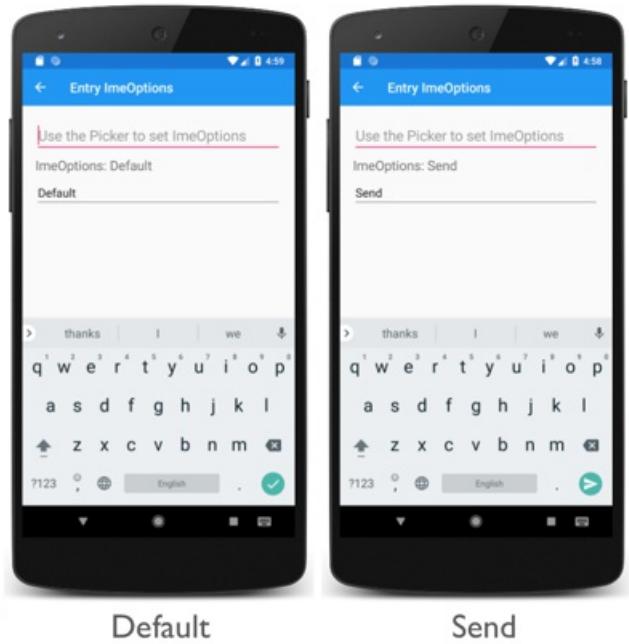
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
entry.On<Android>().SetImeOptions(ImeFlags.Send);
```

The `Entry.On<Android>` method specifies that this platform-specific will only run on Android. The `Entry.SetImeOptions` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to set the input method action option for the soft keyboard for the `Entry`, with the `ImeFlags` enumeration providing the following values:

- `Default` – indicates that no specific action key is required, and that the underlying control will produce its own if it can. This will either be `Next` or `Done`.
- `None` – indicates that no action key will be made available.
- `Go` – indicates that the action key will perform a "go" operation, taking the user to the target of the text they typed.
- `Search` – indicates that the action key performs a "search" operation, taking the user to the results of searching for the text they have typed.
- `Send` – indicates that the action key will perform a "send" operation, delivering the text to its target.
- `Next` – indicates that the action key will perform a "next" operation, taking the user to the next field that will accept text.
- `Done` – indicates that the action key will perform a "done" operation, closing the soft keyboard.
- `Previous` – indicates that the action key will perform a "previous" operation, taking the user to the previous field that will accept text.
- `ImeMaskAction` – the mask to select action options.
- `NoPersonalizedLearning` – indicates that the spellchecker will neither learn from the user, nor suggest corrections based on what the user has previously typed.
- `NoFullscreen` – indicates that the UI should not go fullscreen.
- `NoExtractUi` – indicates that no UI will be shown for extracted text.
- `NoAccessoryAction` – indicates that no UI will be displayed for custom actions.

The result is that a specified `ImeFlags` value is applied to the soft keyboard for the `Entry`, which sets the input method editor options:



Default

Send

### Enabling a Drop Shadow on a ImageButton

This platform-specific is used to enable a drop shadow on a `ImageButton`. It's consumed in XAML by setting the `ImageButton.IsEnabled` bindable property to `true`, along with a number of additional optional bindable properties that control the drop shadow:

```
<ContentPage ...>
    < xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout Margin="20">
            <ImageButton ...
                Source="XamarinLogo.png"
                BackgroundColor="GhostWhite"
                android:ImageButton.IsEnabled="true"
                android:ImageButton.ShadowColor="Gray"
                android:ImageButton.ShadowRadius="12">
                <android:ImageButton.ShadowOffset>
                    <Size>
                        <x:Arguments>
                            <x:Double>10</x:Double>
                            <x:Double>10</x:Double>
                        </x:Arguments>
                    </Size>
                </android:ImageButton.ShadowOffset>
            </ImageButton>
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

var imageView = new Xamarin.Forms.ImageButton { Source = "XamarinLogo.png", BackgroundColor =
Color.GhostWhite, ... };
imageView.On<Android>()
    .SetIsShadowEnabled(true)
    .SetShadowColor(Color.Gray)
    .SetShadowOffset(new Size(10, 10))
    .SetShadowRadius(12);
```

## IMPORTANT

A drop shadow is drawn as part of the `ImageButton` background, and the background is only drawn if the `BackgroundColor` property is set. Therefore, a drop shadow will not be drawn if the `ImageButton.BackgroundColor` property isn't set.

The `ImageButton.On<Android>` method specifies that this platform-specific will only run on Android. The `ImageButton.SetIsShadowEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to control whether a drop shadow is enabled on the `ImageButton`. In addition, the following methods can be invoked to control the drop shadow:

- `SetShadowColor` – sets the color of the drop shadow. The default color is `Color.Default`.
- `SetShadowOffset` – sets the offset of the drop shadow. The offset changes the direction the shadow is cast, and is specified as a `Size` value. The `Size` structure values are expressed in device-independent units, with the first value being the distance to the left (negative value) or right (positive value), and the second value being the distance above (negative value) or below (positive value). The default value of this property is (0.0, 0.0), which results in the shadow being cast around every side of the `ImageButton`.
- `SetShadowRadius` – sets the blur radius used to render the drop shadow. The default radius value is 10.0.

## NOTE

The state of a drop shadow can be queried by calling the `GetIsShadowEnabled`, `GetShadowColor`, `GetShadowOffset`, and `GetShadowRadius` methods.

The result is that a drop shadow can be enabled on a `ImageButton`:



## Enabling Fast Scrolling in a ListView

This platform-specific is used to enable fast scrolling through data in a `ListView`. It's consumed in XAML by setting the `ListView.IsFastScrollEnabled` attached property to a `boolean` value:

```
<ContentPage ...>
    xmlns:android="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        ...
        <ListView ItemsSource="{Binding GroupedEmployees}"
                  GroupDisplayBinding="{Binding Key}"
                  IsGroupingEnabled="true"
                  android:ListView.IsFastScrollEnabled="true">
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```

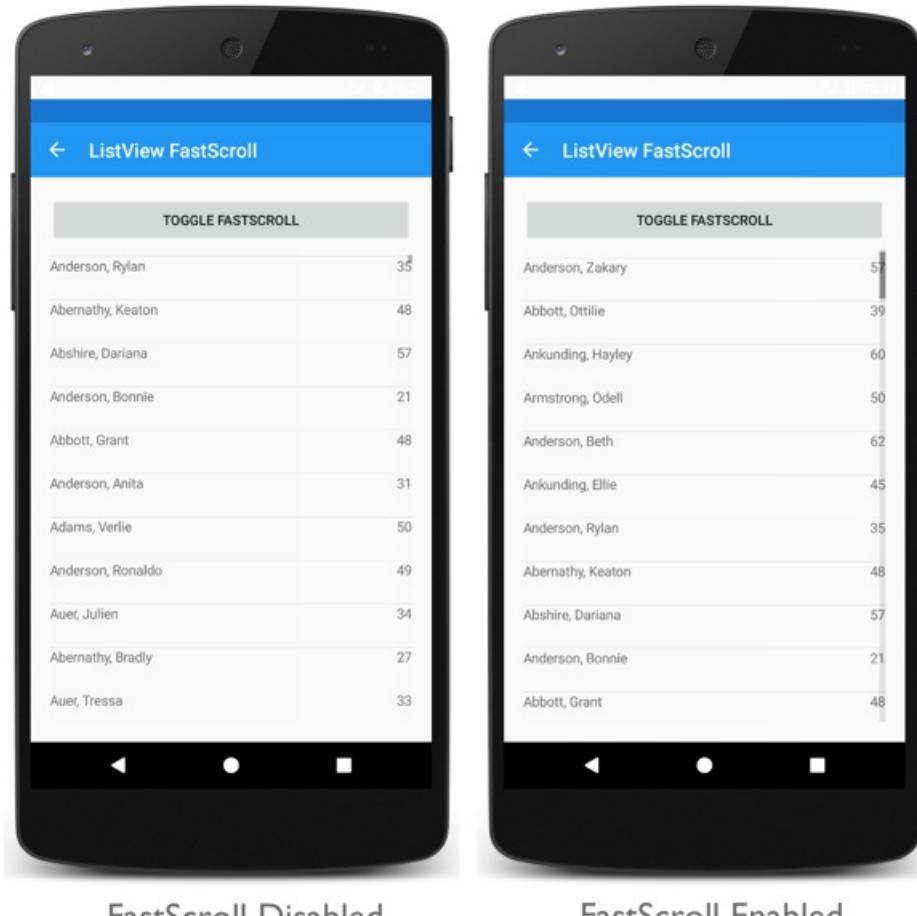
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
var listView = new Xamarin.Forms.ListView { IsGroupingEnabled = true, ... };
listView.SetBinding(ItemsView.ItemsSourceProperty, "GroupedEmployees");
listView.GroupDisplayBinding = new Binding("Key");
listView.On<Android>().SetIsFastScrollEnabled(true);

```

The `ListView.On<Android>` method specifies that this platform-specific will only run on Android. The `ListView.SetIsFastScrollEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to enable fast scrolling through data in a `ListView`. In addition, the `SetIsFastScrollEnabled` method can be used to toggle fast scrolling by calling the `IsFastScrollEnabled` method to return whether fast scrolling is enabled:

```
listView.On<Android>().SetIsFastScrollEnabled(!listView.On<Android>().IsFastScrollEnabled());
```

The result is that fast scrolling through data in a `ListView` can be enabled, which changes the size of the scroll thumb:



### Enabling Mixed Content in a WebView

This platform-specific controls whether a `WebView` can display mixed content in applications that target API 21 or greater. Mixed content is content that's initially loaded over an HTTPS connection, but which loads resources (such as images, audio, video, stylesheets, scripts) over an HTTP connection. It's consumed in XAML by setting the

`WebView.MixedContentMode` attached property to a value of the `MixedContentHandling` enumeration:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <WebView ... android:WebView.MixedContentMode="AlwaysAllow" />
    </ContentPage>
```

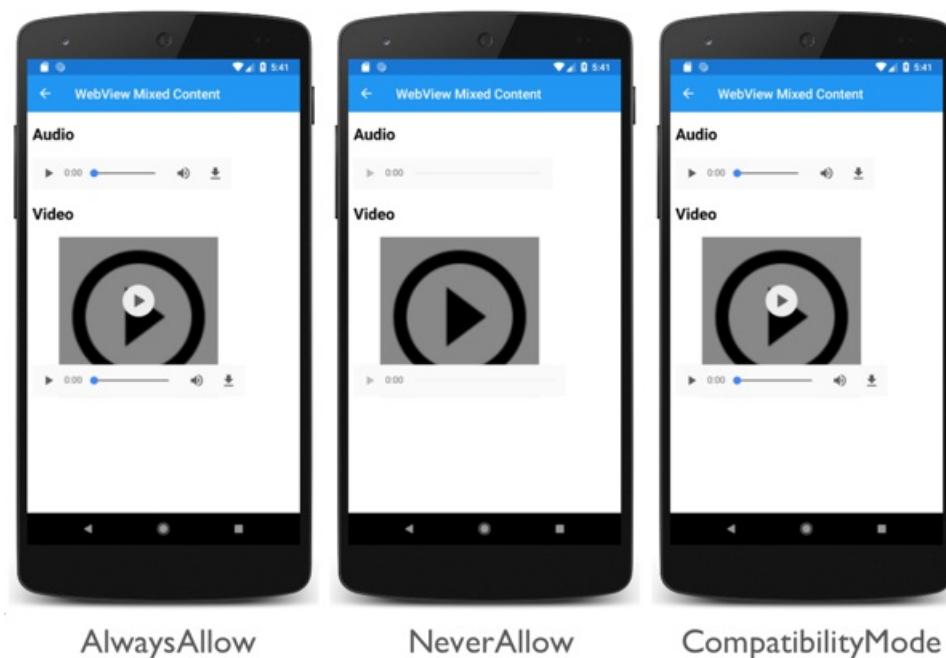
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
webView.On<Android>().SetMixedContentMode(MixedContentHandling.AlwaysAllow);
```

The `WebView.On<Android>` method specifies that this platform-specific will only run on Android. The `WebView.SetMixedContentMode` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to control whether mixed content can be displayed, with the `MixedContentHandling` enumeration providing three possible values:

- `AlwaysAllow` – indicates that the `WebView` will allow an HTTPS origin to load content from an HTTP origin.
- `NeverAllow` – indicates that the `WebView` will not allow an HTTPS origin to load content from an HTTP origin.
- `CompatibilityMode` – indicates that the `WebView` will attempt to be compatible with the approach of the latest device web browser. Some HTTP content may be allowed to be loaded by an HTTPS origin and other types of content will be blocked. The types of content that are blocked or allowed may change with each operating system release.

The result is that a specified `MixedContentHandling` value is applied to the `WebView`, which controls whether mixed content can be displayed:



## Pages

On Android, the following platform-specific functionality is provided for `Xamarin.Forms` pages:

- Setting the height of the navigation bar on a `NavigationPage`. For more information, see [Setting the Navigation Bar Height on a NavigationPage](#).
- Disabling transition animations when navigating through pages in a `TabbedPage`. For more information, see [Disabling Transition Animations](#).

## Disabling Page Transition Animations in a TabbedPage.

- Enabling swiping between pages in a `TabPage`. For more information, see [Enabling Swiping Between Pages in a TabbedPage](#).
- Setting the toolbar placement and color on a `TabPage`. For more information, see [Setting TabbedPage Toolbar Placement and Color](#).

## Setting the Navigation Bar Height on a NavigationPage

This platform-specific sets the height of the navigation bar on a `NavigationPage`. It's consumed in XAML by setting the `NavigationPage.BarHeight` bindable property to an integer value:

```
<NavigationPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;assembly=Xamarin.Forms.Core"  
        android:NavigationBar.BarHeight="450">  
    ...  
</NavigationPage>
```

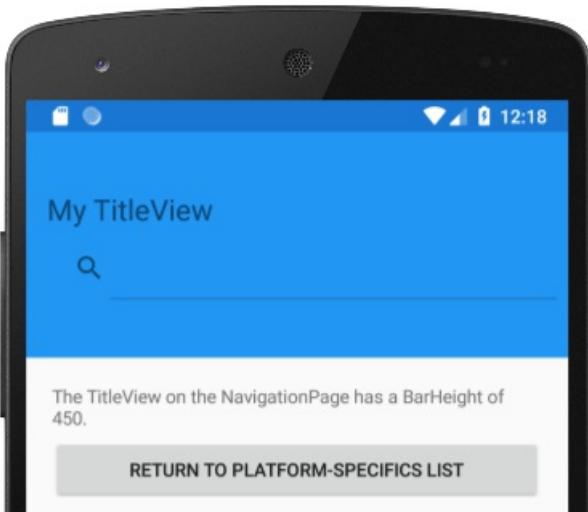
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;  
...  
  
public class AndroidNavigationPageCS : Xamarin.Forms.NavigationPage  
{  
    public AndroidNavigationPageCS()  
    {  
        On<Android>().SetBarHeight(450);  
    }  
}
```

The `NavigationPage.On<Android>` method specifies that this platform-specific will only run on app compat Android.

The `NavigationPage.SetBarHeight` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` namespace, is used to set the height of the navigation bar on a `NavigationPage`. In addition, the `NavigationPage.GetBarHeight` method can be used to return the height of the navigation bar in the `NavigationPage`.

The result is that the height of the navigation bar on a `NavigationPage` can be set:



## Disabling Page Transition Animations in a TabbedPage

This platform-specific is used to disable transition animations when navigating through pages, either programmatically or when using the tab bar, in a `TabPage`. It's consumed in XAML by setting the

`TabPage.IsSmoothScrollEnabled` bindable property to `false`:

```
<TabbedPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
    android:TabbedPage.IsSmoothScrollEnabled="false">  
    ...  
</TabbedPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;  
...  
On<Android>().SetIsSmoothScrollEnabled(false);
```

The `TabPage.On<Android>` method specifies that this platform-specific will only run on Android. The `TabPage.SetIsSmoothScrollEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to control whether transition animations will be displayed when navigating between pages in a `TabbedPage`. In addition, the `TabPage` class in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace also has the following methods:

- `IsSmoothScrollEnabled`, which is used to retrieve whether transition animations will be displayed when navigating between pages in a `TabPage`.
- `EnableSmoothScroll`, which is used to enable transition animations when navigating between pages in a `TabPage`.
- `DisableSmoothScroll`, which is used to disable transition animations when navigating between pages in a `TabPage`.

### Enabling Swiping Between Pages in a TabbedPage

This platform-specific is used to enable swiping with a horizontal finger gesture between pages in a `TabPage`. It's consumed in XAML by setting the `TabPage.IsEnabled` attached property to a `boolean` value:

```
<TabbedPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
    android:TabbedPage.OffscreenPageLimit="2"  
    android:TabbedPage.IsEnabled="true">  
    ...  
</TabbedPage>
```

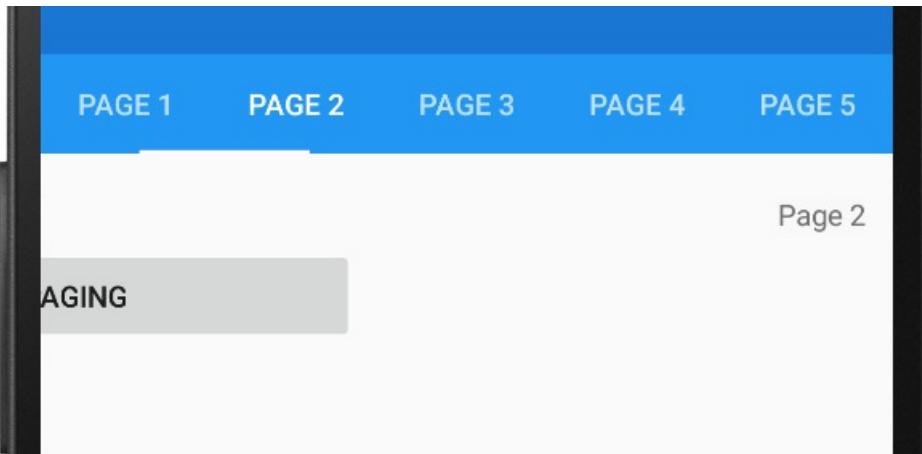
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;  
...  
On<Android>().SetOffscreenPageLimit(2)  
    .SetIsSwipePagingEnabled(true);
```

The `TabPage.On<Android>` method specifies that this platform-specific will only run on Android. The `TabPage.SetIsSwipePagingEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to enable swiping between pages in a `TabPage`. In addition, the `TabPage` class in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace also has a `EnableSwipePaging` method that

enables this platform-specific, and a `DisableSwipePaging` method that disables this platform-specific. The `TabPage.OffscreenPageLimit` attached property, and `SetOffscreenPageLimit` method, are used to set the number of pages that should be retained in an idle state on either side of the current page.

The result is that swipe paging through the pages displayed by a `TabPage` is enabled:



### Setting TabbedPage Toolbar Placement and Color

These platform-specifics are used to set the placement and color of the toolbar on a `TabPage`. They are consumed in XAML by setting the `TabPage.ToolbarPlacement` attached property to a value of the `ToolbarPlacement` enumeration, and the `TabPage.BarItemColor` and `TabPage.BarSelectedItemColor` attached properties to a `Color`:

```
<TabbedPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
        android:TabbedPane.ToolbarPlacement="Bottom"  
        android:TabbedPane.BarItemColor="Black"  
        android:TabbedPane.BarSelectedItemColor="Red">  
    ...  
</TabbedPage>
```

Alternatively, they can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;  
...  
  
On<Android>().SetToolbarPlacement(ToolbarPlacement.Bottom)  
    .SetBarItemColor(Color.Black)  
    .SetBarSelectedItemColor(Color.Red);
```

The `TabPage.On<Android>` method specifies that these platform-specifics will only run on Android. The `TabPage.SetToolbarPlacement` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to set the toolbar placement on a `TabPage`, with the `ToolbarPlacement` enumeration providing the following values:

- `Default` – indicates that the toolbar is placed at the default location on the page. This is the top of the page on phones, and the bottom of the page on other device idioms.
- `Top` – indicates that the toolbar is placed at the top of the page.
- `Bottom` – indicates that the toolbar is placed at the bottom of the page.

In addition, the `TabPage.SetBarItemColor` and `TabPage.SetBarSelectedItemColor` methods are used to set the color of toolbar items and selected toolbar items, respectively.

## NOTE

The `GetToolbarPlacement`, `GetBarItemColor`, and `GetBarSelectedItemColor` methods can be used to retrieve the placement and color of the `TabbedPage` toolbar.

The result is that the toolbar placement, the color of toolbar items, and the color of the selected toolbar item can be set on a `TabbedPage`:



## Application

On Android, the following platform-specific functionality is provided for the `Xamarin.Forms Application` class:

- Setting the operating mode of a soft keyboard. For more information, see [Setting the Soft Keyboard Input Mode](#).
- Disabling the `Disappearing` and `Appearing` page lifecycle events on pause and resume respectively, for applications that use AppCompat. For more information, see [Disabling the Disappearing and Appearing Page Lifecycle Events](#).

### Setting the Soft Keyboard Input Mode

This platform-specific is used to set the operating mode for a soft keyboard input area, and is consumed in XAML by setting the `Application.WindowSoftInputModeAdjust` attached property to a value of the

`WindowSoftInputModeAdjust` enumeration:

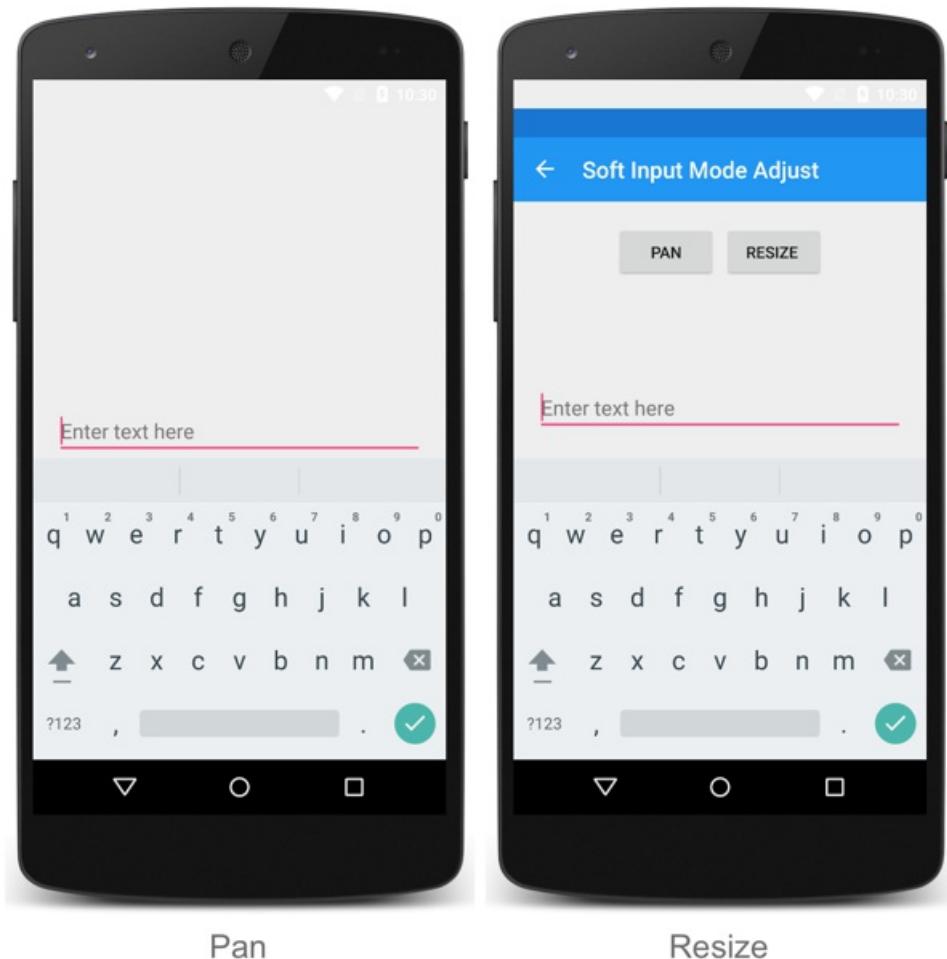
```
<Application ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
    android:Application.WindowSoftInputModeAdjust="Resize">  
    ...  
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;  
...  
  
App.Current.On<Android>().UseWindowSoftInputModeAdjust(WindowSoftInputModeAdjust.Resize);
```

The `Application.On<Android>` method specifies that this platform-specific will only run on Android. The `Application.UseWindowSoftInputModeAdjust` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to set the soft keyboard input area operating mode, with the `WindowSoftInputModeAdjust` enumeration providing two values: `Pan` and `Resize`. The `Pan` value uses the `AdjustPan` adjustment option, which doesn't resize the window when an input control has focus. Instead, the contents of the window are panned so that the current focus isn't obscured by the soft keyboard. The `Resize` value uses the `AdjustResize` adjustment option, which resizes the window when an input control has focus, to make room for the soft keyboard.

The result is that the soft keyboard input area operating mode can be set when an input control has focus:



Pan

Resize

### Disabling the Disappearing and Appearing Page Lifecycle Events

This platform-specific is used to disable the `Disappearing` and `Appearing` page events on application pause and resume respectively, for applications that use AppCompat. In addition, it includes the ability to control whether the soft keyboard is displayed on resume, if it was displayed on pause, provided that the operating mode of the soft keyboard is set to `WindowSoftInputModeAdjust.Resize`.

#### NOTE

Note that these events are enabled by default to preserve existing behavior for applications that rely on the events. Disabling these events makes the AppCompat event cycle match the pre-AppCompat event cycle.

This platform-specific can be consumed in XAML by setting the `Application.SendDisappearingEventOnPause`, `Application.SendAppearingEventOnResume`, and `Application.ShouldPreserveKeyboardOnResume` attached properties to `boolean` values:

```
<Application ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
    xmlns:androidAppCompat="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;assembly=Xamarin.Forms.Core"  
        android:Application.WindowSoftInputModeAdjust="Resize"  
        androidAppCompat:Application.SendDisappearingEventOnPause="false"  
        androidAppCompat:Application.SendAppearingEventOnResume="false"  
        androidAppCompat:Application.ShouldPreserveKeyboardOnResume="true">  
    ...  
</Application>
```

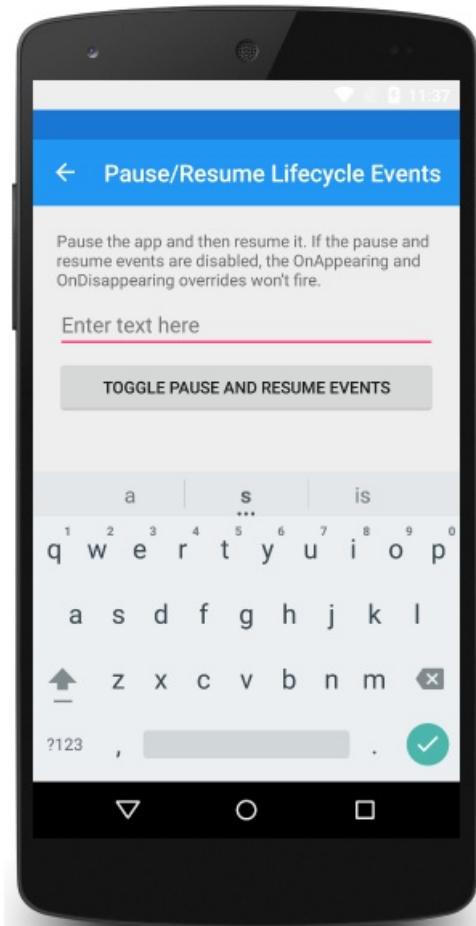
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;
...

Xamarin.Forms.Application.Current.On<Android>()
    .UseWindowSoftInputModeAdjust(WindowSoftInputModeAdjust.Resize)
    .SendDisappearingEventOnPause(false)
    .SendAppearingEventOnResume(false)
    .ShouldPreserveKeyboardOnResume(true);
```

The `Application.Current.On<Android>` method specifies that this platform-specific will only run on Android. The `Application.SendDisappearingEventOnPause` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` namespace, is used to enable or disable firing the `Disappearing` page event, when the application enters the background. The `Application.SendAppearingEventOnResume` method is used to enable or disable firing the `Appearing` page event, when the application resumes from the background. The `Application.ShouldPreserveKeyboardOnResume` method is used control whether the soft keyboard is displayed on resume, if it was displayed on pause, provided that the operating mode of the soft keyboard is set to `WindowSoftInputModeAdjust.Resize`.

The result is that the `Disappearing` and `Appearing` page events won't be fired on application pause and resume respectively, and that if the soft keyboard was displayed when the application was paused, it will also be displayed when the application resumes:



## Summary

This article demonstrated how to consume the Android platform-specifics that are built into Xamarin.Forms.

Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects.

## Related Links

- [Creating Platform-Specifics](#)
- [PlatformSpecifics \(sample\)](#)
- [AndroidSpecific](#)
- [AndroidSpecific.AppCompat](#)

# Windows Platform-Specifics

9/20/2018 • 12 minutes to read • [Edit Online](#)

*Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects. This article demonstrates how to consume the Windows platform-specifics that are built into Xamarin.Forms.*

## VisualElements

On the Universal Windows Platform, the following platform-specific functionality is provided for Xamarin.Forms views, pages, and layouts:

- Setting an access key for a `VisualElement`. For more information, see [Setting VisualElement Access Keys](#).
- Disabling legacy color mode on a supported `VisualElement`. For more information, see [Disabling Legacy Color Mode](#).

### Setting VisualElement Access Keys

Access keys are keyboard shortcuts that improve the usability and accessibility of apps on the Universal Windows Platform by providing an intuitive way for users to quickly navigate and interact with the app's visible UI through a keyboard instead of via touch or a mouse. They are combinations of the Alt key and one or more alphanumeric keys, typically pressed sequentially. Keyboard shortcuts are automatically supported for access keys that use a single alphanumeric character.

Access key tips are floating badges displayed next to controls that include access keys. Each access key tip contains the alphanumeric keys that activate the associated control. When a user presses the Alt key, the access key tips are displayed.

This platform-specific is used to specify an access key for a `VisualElement`. It's consumed in XAML by setting the `VisualElement.AccessKey` attached property to an alphanumeric value, and by optionally setting the `VisualElement.AccessKeyPlacement` attached property to a value of the `AccessKeyPlacement` enumeration, the `VisualElement.AccessKeyHorizontalOffset` attached property to a `double`, and the `VisualElement.AccessKeyVerticalOffset` attached property to a `double`:

```

<TabbedPage ...>
    <ContentPage Title="Page 1">
        <StackLayout Margin="20">
            ...
            <Switch windows:VisualElement.AccessKey="A" />
            <Entry Placeholder="Enter text here"
                  windows:VisualElement.AccessKey="B" />
            ...
            <Button Text="Access key F, placement top with offsets"
                  Margin="20"
                  Clicked="OnButtonClicked"
                  windows:VisualElement.AccessKey="F"
                  windows:VisualElement.AccessKeyPlacement="Top"
                  windows:VisualElement.AccessKeyHorizontalOffset="20"
                  windows:VisualElement.AccessKeyVerticalOffset="20" />
            ...
        </StackLayout>
    </ContentPage>
    ...
</TabbedPage>

```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

var page = new ContentPage { Title = "Page 1" };
page.On<Windows>().SetAccessKey("1");

var switchView = new Switch();
switchView.On<Windows>().SetAccessKey("A");
var entry = new Entry { Placeholder = "Enter text here" };
entry.On<Windows>().SetAccessKey("B");
...

var button4 = new Button { Text = "Access key F, placement top with offsets", Margin = new Thickness(20) };
button4.Clicked += OnButtonClicked;
button4.On<Windows>()
    .SetAccessKey("F")
    .SetAccessKeyPlacement(AccessKeyPlacement.Top)
    .SetAccessKeyHorizontalOffset(20)
    .SetAccessKeyVerticalOffset(20);
...

```

The `VisualElement.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `VisualElement.SetAccessKey` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to set the access key value for the `VisualElement`. The `VisualElement.SetAccessKeyPlacement` method, optionally specifies the position to use for displaying the access key tip, with the `AccessKeyPlacement` enumeration providing the following possible values:

- `Auto` – indicates that the access key tip placement will be determined by the operating system.
- `Top` – indicates that the access key tip will appear above the top edge of the `VisualElement`.
- `Bottom` – indicates that the access key tip will appear below the lower edge of the `VisualElement`.
- `Right` – indicates that the access key tip will appear to the right of the right edge of the `VisualElement`.
- `Left` – indicates that the access key tip will appear to the left of the left edge of the `VisualElement`.
- `Center` – indicates that the access key tip will appear overlaid on the center of the `VisualElement`.

#### NOTE

Typically, the `Auto` key tip placement is sufficient, which includes support for adaptive user interfaces.

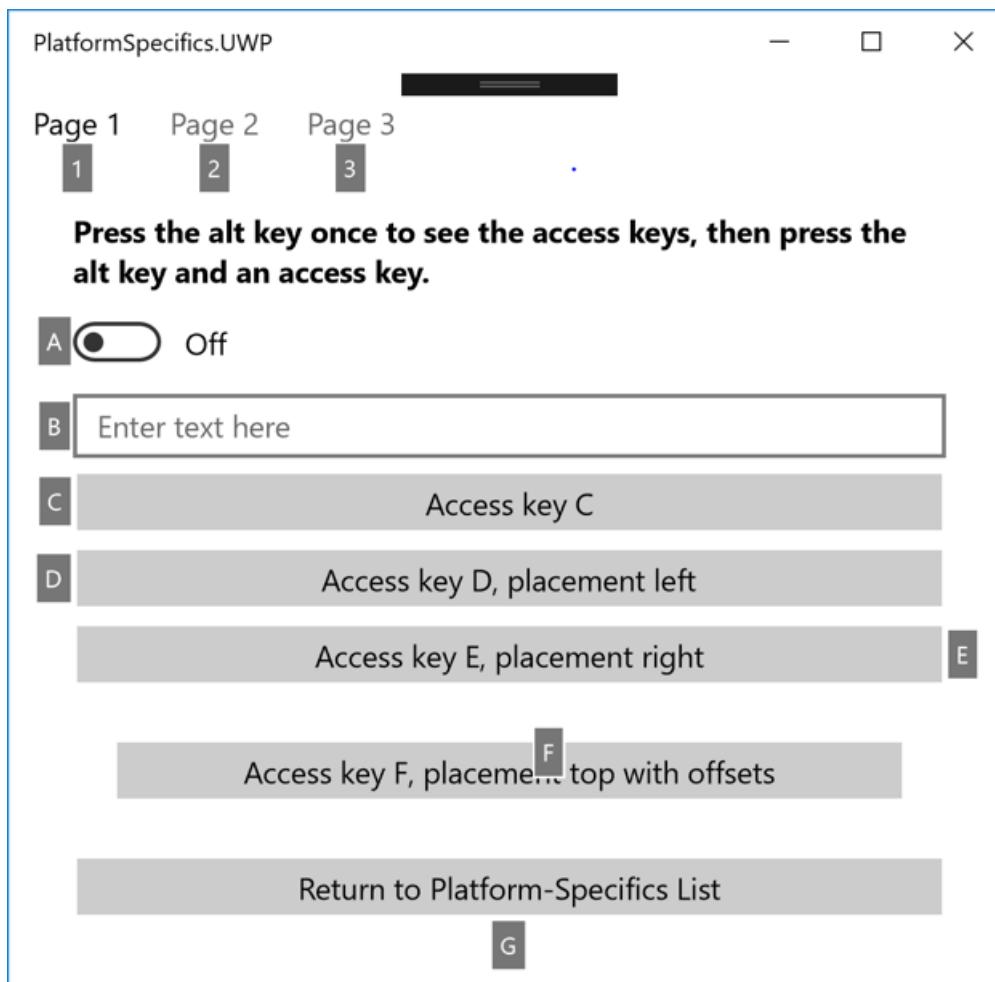
The `VisualElement.SetAccessKeyHorizontalOffset` and `VisualElement.SetAccessKeyVerticalOffset` methods can be used for more granular control of the access key tip location. The argument to the `SetAccessKeyHorizontalOffset` method indicates how far to move the access key tip left or right, and the argument to the `SetAccessKeyVerticalOffset` method indicates how far to move the access key tip up or down.

#### NOTE

Access key tip offsets can't be set when the access key placement is set `Auto`.

In addition, the `GetAccessKey`, `GetAccessKeyPlacement`, `GetAccessKeyHorizontalOffset`, and `GetAccessKeyVerticalOffset` methods can be used to retrieve an access key value and its location.

The result is that access key tips can be displayed next to any `VisualElement` instances that define access keys, by pressing the Alt key:



When a user activates an access key, by pressing the Alt key followed by the access key, the default action for the `VisualElement` will be executed. For example, when a user activates the access key on a `Switch`, the `Switch` is toggled. When a user activates the access key on an `Entry`, the `Entry` gains focus. When a user activates the access key on a `Button`, the event handler for the `Clicked` event is executed.

For more information about access keys, see [Access keys](#).

#### Disabling Legacy Color Mode

Some of the Xamarin.Forms views feature a legacy color mode. In this mode, when the `IsEnabled` property of the view is set to `false`, the view will override the colors set by the user with the default native colors for the disabled state. For backwards compatibility, this legacy color mode remains the default behavior for supported views.

This platform-specific disables this legacy color mode, so that colors set on a view by the user remain even when the view is disabled. It's consumed in XAML by setting the `VisualElement.IsEnabled` attached property to `false`:

```
<ContentPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            ...
            <Editor Text="Enter text here"
                    TextColor="Blue"
                    BackgroundColor="Bisque"
                    windows:VisualElement.IsEnabled="False" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
_legacyColorModeDisabledEditor.On<Windows>().SetIsLegacyColorModeEnabled(false);
```

The `VisualElement.On<Windows>` method specifies that this platform-specific will only run on Windows. The `VisualElement.SetIsLegacyColorModeEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to control whether the legacy color mode is disabled. In addition, the `VisualElement.GetIsLegacyColorModeEnabled` method can be used to return whether the legacy color mode is disabled.

The result is that the legacy color mode can be disabled, so that colors set on a view by the user remain even when the view is disabled:

The Editor below uses the legacy color mode. When `IsEnabled` is `false`, it uses the default native colors for the control.

Enter text here

Toggle IsEnabled (Currently: False)

The Editor below has the legacy color mode disabled. It will use whatever colors are manually set.

Enter text here

Toggle IsEnabled (Currently: False)

## NOTE

When setting a `VisualStateGroup` on a view, the legacy color mode is completely ignored. For more information about visual states, see [The Xamarin.Forms Visual State Manager](#).

# Views

On the Universal Windows Platform (UWP), the following platform-specific functionality is provided for Xamarin.Forms views:

- Detecting reading order from text content in `Entry`, `Editor`, and `Label` instances. For more information, see [Detecting Reading Order from Content](#).
- Enabling tap gesture support in a `ListView`. For more information, see [Enabling Tap Gesture Support in a ListView](#).
- Enabling a `SearchBar` to interact with the spell check engine. For more information, see [Enabling SearchBar Spell Check](#).
- Enabling a `WebView` to display JavaScript alerts in a UWP message dialog. For more information, see [Displaying JavaScript Alerts](#).

## Detecting Reading Order from Content

This platform-specific enables the reading order (left-to-right or right-to-left) of bidirectional text in `Entry`, `Editor`, and `Label` instances to be detected dynamically. It's consumed in XAML by setting the `InputView.DetectReadingOrderFromContent` (for `Entry` and `Editor` instances) or `Label.DetectReadingOrderFromContent` attached property to a `boolean` value:

```
<ContentPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            <Editor ... windows:InputView.DetectReadingOrderFromContent="true" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

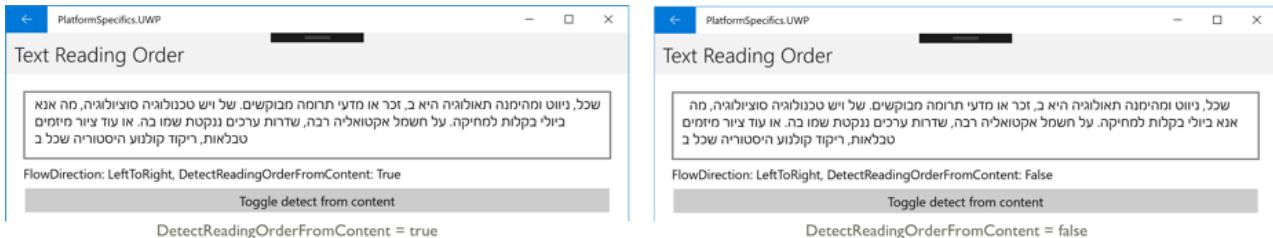
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
editor.On<Windows>().SetDetectReadingOrderFromContent(true);
```

The `Editor.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `InputView.SetDetectReadingOrderFromContent` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to control whether the reading order is detected from the content in the `InputView`. In addition, the `InputView.SetDetectReadingOrderFromContent` method can be used to toggle whether the reading order is detected from the content by calling the `InputView.GetDetectReadingOrderFromContent` method to return the current value:

```
editor.On<Windows>().SetDetectReadingOrderFromContent(!editor.On<Windows>()
    .GetDetectReadingOrderFromContent());
```

The result is that `Entry`, `Editor`, and `Label` instances can have the reading order of their content detected

dynamically:



#### NOTE

Unlike setting the `FlowDirection` property, the logic for views that detect the reading order from their text content will not affect the alignment of text within the view. Instead, it adjusts the order in which blocks of bidirectional text are laid out.

### Enabling Tap Gesture Support in a ListView

On the Universal Windows Platform, by default the `Xamarin.Forms.ListView` uses the native `ItemClick` event to respond to interaction, rather than the native `Tapped` event. This provides accessibility functionality so that the Windows Narrator and the keyboard can interact with the `ListView`. However, it also renders any tap gestures inside the `ListView` inoperable.

This platform-specific controls whether items in a `ListView` can respond to tap gestures, and hence whether the native `ListView` fires the `ItemClick` or `Tapped` event. It's consumed in XAML by setting the `ListView.SelectionMode` attached property to a value of the `ListViewSelectionMode` enumeration:

```
<ContentPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            <ListView ... windows:ListView.SelectionMode="Inaccessible">
                ...
            </ListView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
listView.On<Windows>().SetSelectionMode(ListViewSelectionMode.Inaccessible);
```

The `ListView.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `ListView.SetSelectionMode` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to control whether items in a `ListView` can respond to tap gestures, with the `ListViewSelectionMode` enumeration providing two possible values:

- `Accessible` – indicates that the `ListView` will fire the native `ItemClick` event to handle interaction, and hence provide accessibility functionality. Therefore, the Windows Narrator and the keyboard can interact with the `ListView`. However, items in the `ListView` can't respond to tap gestures. This is the default behavior for `ListView` instances on the Universal Windows Platform.
- `Inaccessible` – indicates that the `ListView` will fire the native `Tapped` event to handle interaction. Therefore, items in the `ListView` can respond to tap gestures. However, there's no accessibility functionality and hence the Windows Narrator and the keyboard can't interact with the `ListView`.

## NOTE

The `Accessible` and `Inaccessible` selection modes are mutually exclusive, and you will need to choose between an accessible `ListView` or a `ListView` that can respond to tap gestures.

In addition, the `GetSelectionMode` method can be used to return the current `ListViewSelectionMode`.

The result is that a specified `ListViewSelectionMode` is applied to the `ListView`, which controls whether items in the `ListView` can respond to tap gestures, and hence whether the native `ListView` fires the `ItemClick` or `Tapped` event.

## Enabling SearchBar Spell Check

This platform-specific enables a `SearchBar` to interact with the spell check engine. It's consumed in XAML by setting the `SearchBar.IsSpellCheckEnabled` attached property to a `boolean` value:

```
<ContentPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <SearchBar ... windows:SearchBar.IsSpellCheckEnabled="true" />  
        ...  
    </StackLayout>  
</ContentPage>
```

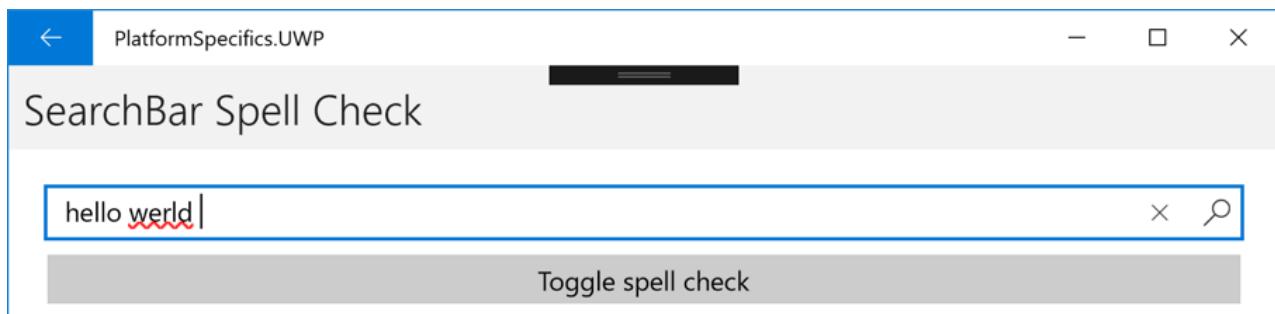
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;  
...  
  
searchBar.On<Windows>().SetIsSpellCheckEnabled(true);
```

The `SearchBar.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `SearchBar.SetIsSpellCheckEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, turns the spell checker on and off. In addition, the `SearchBar.SetIsSpellCheckEnabled` method can be used to toggle the spell checker by calling the `SearchBar.GetIsSpellCheckEnabled` method to return whether the spell checker is enabled:

```
searchBar.On<Windows>().SetIsSpellCheckEnabled(!searchBar.On<Windows>().GetIsSpellCheckEnabled());
```

The result is that text entered into the `SearchBar` can be spell checked, with incorrect spellings being indicated to the user:



## NOTE

The `SearchBar` class in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace also has `EnableSpellCheck` and `DisableSpellCheck` methods that can be used to enable and disable the spell checker on the `SearchBar`, respectively.

## Displaying JavaScript Alerts

This platform-specific enables a `WebView` to display JavaScript alerts in a UWP message dialog. It's consumed in XAML by setting the `WebView.IsJavaScriptAlertEnabled` attached property to a `boolean` value:

```
<ContentPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <WebView ... windows:WebView.IsJavaScriptAlertEnabled="true" />  
        ...  
    </StackLayout>  
</ContentPage>
```

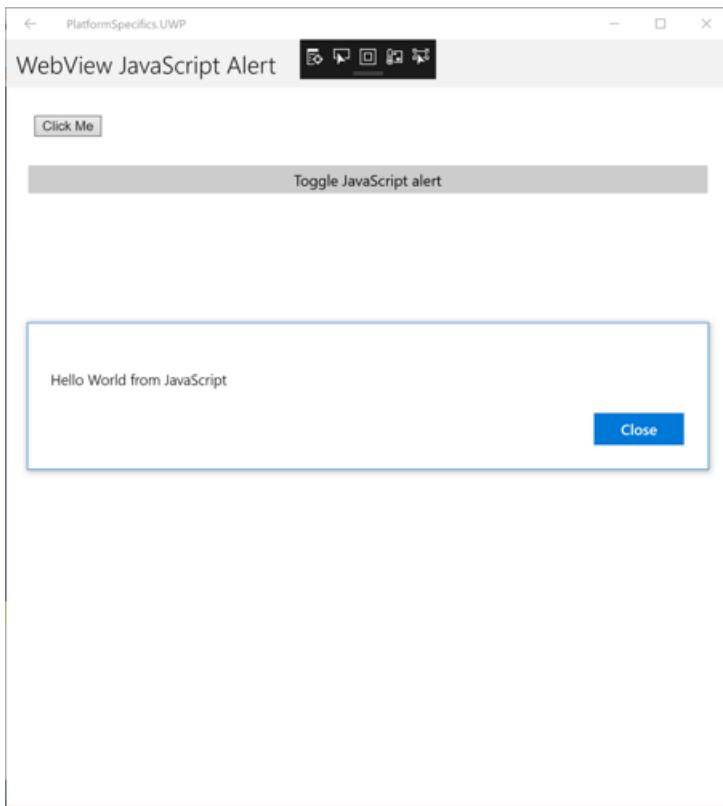
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;  
...  
  
var webView = new Xamarin.Forms.WebView  
{  
    Source = new HtmlWebViewSource  
    {  
        Html = @"<html><body><button onclick=""window.alert('Hello World from JavaScript');"">Click Me</button>  
        </body></html>"  
    }  
};  
webView.On<Windows>().SetIsJavaScriptAlertEnabled(true);
```

The `WebView.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `WebView.SetIsJavaScriptAlertEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to control whether JavaScript alerts are enabled. In addition, the `WebView.SetIsJavaScriptAlertEnabled` method can be used to toggle JavaScript alerts by calling the `IsJavaScriptAlertEnabled` method to return whether they are enabled:

```
_webView.On<Windows>().SetIsJavaScriptAlertEnabled(!_webView.On<Windows>().IsJavaScriptAlertEnabled());
```

The result is that JavaScript alerts can be displayed in a UWP message dialog:



## Pages

On the Universal Windows Platform, the following platform-specific functionality is provided for Xamarin.Forms pages:

- Collapsing the `MasterDetailPage` navigation bar. For more information, see [Collapsing a MasterDetailPage Navigation Bar](#).
- Setting toolbar placement options. For more information, see [Changing the Page Toolbar Placement](#).
- Enabling page icons to be displayed on a `TabbedPage` toolbar. For more information, see [Enabling Icons on a TabbedPage](#).

### Collapsing a MasterDetailPage Navigation Bar

This platform-specific is used to collapse the navigation bar on a `MasterDetailPage`, and is consumed in XAML by setting the `MasterDetailPage.CollapseStyle` and `MasterDetailPage.CollapsedPaneWidth` attached properties:

```
<MasterDetailPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"  
    windows:MasterDetailPage.CollapseStyle="Partial"  
    windows:MasterDetailPage.CollapsedPaneWidth="48">  
    ...  
</MasterDetailPage>
```

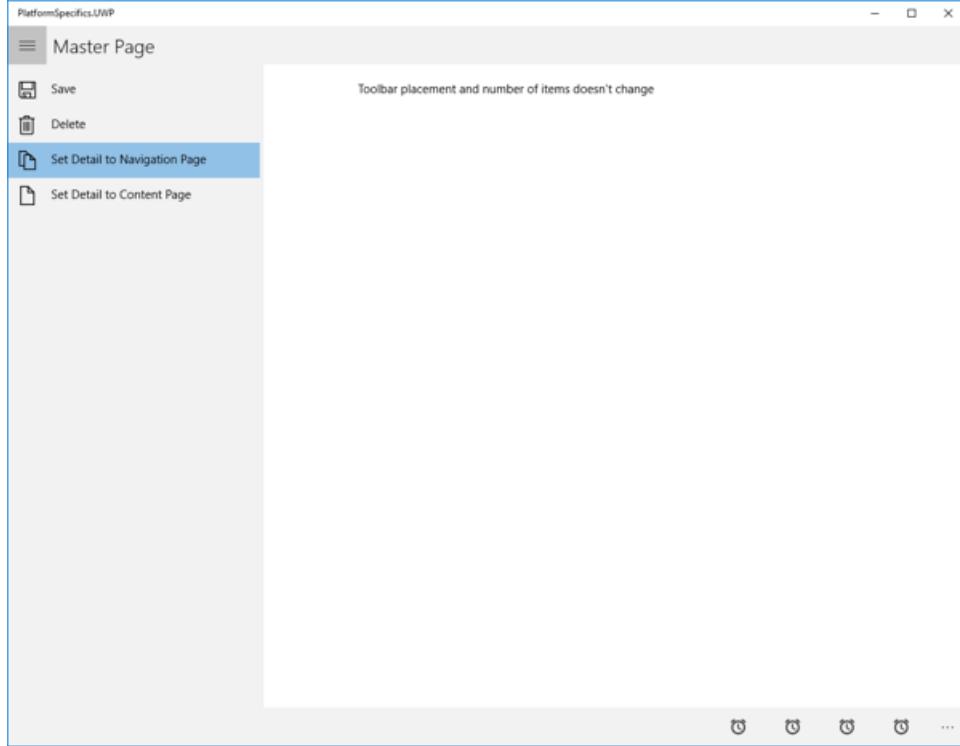
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;  
...  
page.On<Windows>().SetCollapseStyle(CollapseStyle.Partial).CollapsedPaneWidth(148);
```

The `MasterDetailPage.On<Windows>` method specifies that this platform-specific will only run on Windows. The

`Page.SetCollapseStyle` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to specify the collapse style, with the `CollapseStyle` enumeration providing two values: `Full` and `Partial`. The `MasterDetailPage.CollapsedPaneWidth` method is used to specify the width of a partially collapsed navigation bar.

The result is that a specified `CollapseStyle` is applied to the `MasterDetailPage` instance, with the width also being specified:



## Changing the Page Toolbar Placement

This platform-specific is used to change the placement of a toolbar on a `Page`, and is consumed in XAML by setting the `Page.ToolbarPlacement` attached property to a value of the `ToolbarPlacement` enumeration:

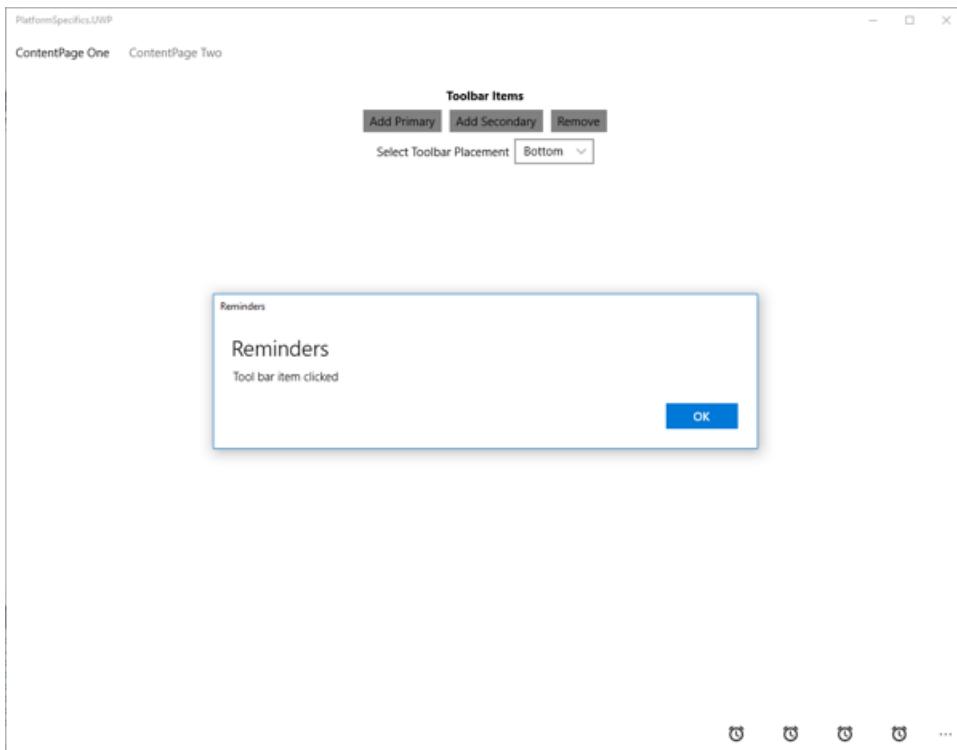
```
<TabbedPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"  
    windows:Page.ToolbarPlacement="Bottom">  
    ...  
</TabbedPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;  
...  
page.On<Windows>().SetToolbarPlacement(ToolbarPlacement.Bottom);
```

The `Page.On<Windows>` method specifies that this platform-specific will only run on Windows. The `Page.SetToolbarPlacement` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to set the toolbar placement, with the `ToolbarPlacement` enumeration providing three values: `Default`, `Top`, and `Bottom`.

The result is that the specified toolbar placement is applied to the `Page` instance:



## Enabling Icons on a TabbedPage

This platform-specific enables page icons to be displayed on a `TabbedPage` toolbar, and provides the ability to optionally specify the icon size. It's consumed in XAML by setting the `TabbedPage.HeaderIconsEnabled` attached property to `true`, and by optionally setting the `TabbedPage.HeaderIconSize` attached property to a `Size` value:

```
<TabbedPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
    windows:TabbedPage.HeaderIconsEnabled="true">
        <windows:TabbedPage.HeaderIconSize>
            <Size>
                <x:Arguments>
                    <x:Double>24</x:Double>
                    <x:Double>24</x:Double>
                </x:Arguments>
            </Size>
        </windows:TabbedPage.HeaderIconSize>
        <ContentPage Title="Todo" Icon="todo.png">
            ...
        </ContentPage>
        <ContentPage Title="Reminders" Icon="reminders.png">
            ...
        </ContentPage>
        <ContentPage Title="Contacts" Icon="contacts.png">
            ...
        </ContentPage>
    </TabbedPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```

using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
public class WindowsTabbedPageIconsCS : Xamarin.Forms.TabbedPage
{
    public WindowsTabbedPageIconsCS()
    {
        On<Windows>().SetHeaderIconsEnabled(true);
        On<Windows>().SetHeaderIconsSize(new Size(24, 24));

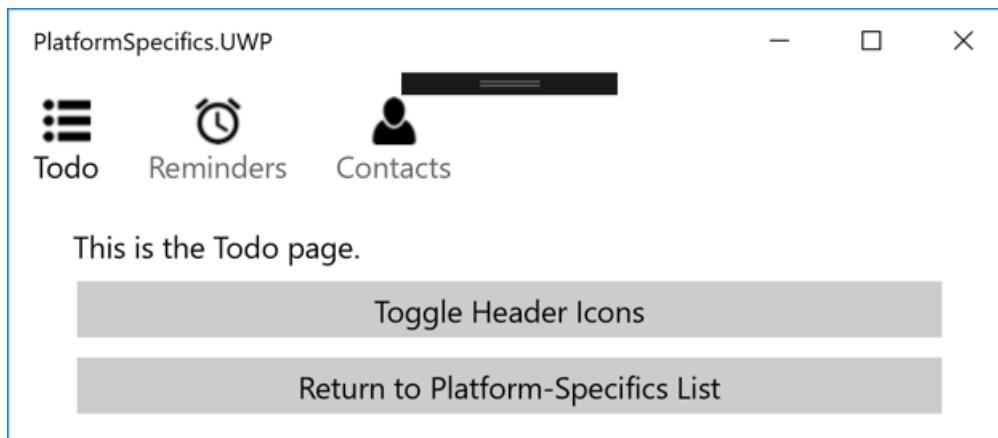
        Children.Add(new ContentPage { Title = "Todo", Icon = "todo.png" });
        Children.Add(new ContentPage { Title = "Reminders", Icon = "reminders.png" });
        Children.Add(new ContentPage { Title = "Contacts", Icon = "contacts.png" });
    }
}

```

The `TabPage.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `TabPage.SetHeaderIconsEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to turn header icons on or off. The `TabPage.SetHeaderIconsSize` method optionally specifies the header icon size with a `Size` value.

In addition, the `TabPage` class in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace also has a `EnableHeaderIcons` method that enables header icons, a `DisableHeaderIcons` method that disables header icons, and a `IsHeaderIconsEnabled` method that returns a `boolean` value that indicates whether header icons are enabled.

The result is that page icons can be displayed on a `TabPage` toolbar, with the icon size being optionally set to a desired size:



## Summary

This article demonstrated how to consume the Windows platform-specifics that are built into Xamarin.Forms. Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects.

## Related Links

- [Creating Platform-Specifics](#)
- [PlatformSpecifics \(sample\)](#)
- [WindowsSpecific](#)

# Creating Platform-Specifics

7/12/2018 • 5 minutes to read • [Edit Online](#)

*Vendors can create their own platform-specifics with Effects. An Effect provides the specific functionality, which is then exposed through a platform-specific. The result is an Effect that can be more easily consumed through XAML, and through a fluent code API. This article demonstrates how to expose an Effect through a platform-specific.*

## Overview

The process for creating a platform-specific is as follows:

1. Implement the specific functionality as an Effect. For more information, see [Creating an Effect](#).
2. Create a platform-specific class that will expose the Effect. For more information, see [Creating a Platform-Specific Class](#).
3. In the platform-specific class, implement an attached property to allow the platform-specific to be consumed through XAML. For more information, see [Adding an Attached Property](#).
4. In the platform-specific class, implement extension methods to allow the platform-specific to be consumed through a fluent code API. For more information, see [Adding Extension Methods](#).
5. Modify the Effect implementation so that the Effect is only applied if the platform-specific has been invoked on the same platform as the Effect. For more information, see [Creating the Effect](#).

The result of exposing an Effect as a platform-specific is that the Effect can be more easily consumed through XAML and through a fluent code API.

### NOTE

It's envisaged that vendors will use this technique to create their own platform-specifics, for ease of consumption by users. While users may choose to create their own platform-specifics, it should be noted that it requires more code than creating and consuming an Effect.

The sample application demonstrates a `Shadow` platform-specific that adds a shadow to the text displayed by a `Label` control:

**Label Shadow Effect**

**Label Shadow Effect**

iOS

Android

The sample application implements the `Shadow` platform-specific on each platform, for ease of understanding. However, aside from each platform-specific Effect implementation, the implementation of the `Shadow` class is largely identical for each platform. Therefore, this guide focusses on the implementation of the `Shadow` class and associated Effect on a single platform.

For more information about Effects, see [Customizing Controls with Effects](#).

## Creating a Platform-Specific Class

A platform-specific is created as a `public static` class:

```
namespace MyCompany.Forms.PlatformConfiguration.iOS
{
    public static Shadow
    {
        ...
    }
}
```

The following sections discuss the implementation of the `Shadow` platform-specific and associated Effect.

## Adding an Attached Property

An attached property must be added to the `Shadow` platform-specific to allow consumption through XAML:

```
namespace MyCompany.Forms.PlatformConfiguration.iOS
{
    using System.Linq;
    using Xamarin.Forms;
    using Xamarin.Forms.PlatformConfiguration;
    using FormsElement = Xamarin.Forms.Label;

    public static class Shadow
    {
        const string EffectName = "MyCompany.LabelShadowEffect";

        public static readonly BindableProperty IsShadowedProperty =
            BindableProperty.CreateAttached("IsShadowed",
                typeof(bool),
                typeof(Shadow),
                false,
                propertyChanged: OnIsShadowedPropertyChanged);

        public static bool GetIsShadowed(BindableObject element)
        {
            return (bool)element.GetValue(IsShadowedProperty);
        }

        public static void SetIsShadowed(BindableObject element, bool value)
        {
            element.SetValue(IsShadowedProperty, value);
        }

        ...

        static void OnIsShadowedPropertyChanged(BindableObject element, object oldValue, object newValue)
        {
            if ((bool)newValue)
            {
                AttachEffect(element as FormsElement);
            }
            else
            {
                DetachEffect(element as FormsElement);
            }
        }

        static void AttachEffect(FormsElement element)
        {
            IElementController controller = element;
            if (controller == null || controller.EffectIsAttached(EffectName))
            {
                return;
            }
            element.Effects.Add(Effect.Resolve(EffectName));
        }
    }
}
```

```

    }

    static void DetachEffect(FormsElement element)
    {
        IElementController controller = element;
        if (controller == null || !controller.EffectIsAttached(EffectName))
        {
            return;
        }

        var toRemove = element.Effects.FirstOrDefault(e => e.ResolveId ==
Effect.Resolve(EffectName).ResolveId);
        if (toRemove != null)
        {
            element.Effects.Remove(toRemove);
        }
    }
}

```

The `IsShadowed` attached property is used to add the `MyCompany.LabelShadowEffect` Effect to, and remove it from, the control that the `Shadow` class is attached to. This attached property registers the `OnIsShadowedPropertyChanged` method that will be executed when the value of the property changes. In turn, this method calls the `AttachEffect` or `DetachEffect` method to add or remove the effect based on the value of the `IsShadowed` attached property. The Effect is added to or removed from the control by modifying the control's `Effects` collection.

#### NOTE

Note that the Effect is resolved by specifying a value that's a concatenation of the resolution group name and unique identifier that's specified on the Effect implementation. For more information, see [Creating an Effect](#).

For more information about attached properties, see [Attached Properties](#).

#### Adding Extension Methods

Extension methods must be added to the `Shadow` platform-specific to allow consumption through a fluent code API:

```

namespace MyCompany.Forms.PlatformConfiguration.iOS
{
    using System.Linq;
    using Xamarin.Forms;
    using Xamarin.Forms.PlatformConfiguration;
    using FormsElement = Xamarin.Forms.Label;

    public static class Shadow
    {
        ...
        public static bool IsShadowed(this IPlatformElementConfiguration<iOS, FormsElement> config)
        {
            return GetIsShadowed(config.Element);
        }

        public static IPlatformElementConfiguration<iOS, FormsElement> SetIsShadowed(this
IPlatformElementConfiguration<iOS, FormsElement> config, bool value)
        {
            SetIsShadowed(config.Element, value);
            return config;
        }
        ...
    }
}

```

The `IsShadowed` and `SetIsShadowed` extension methods invoke the get and set accessors for the `IsShadowed` attached property, respectively. Each extension method operates on the `IPlatformElementConfiguration<iOS, FormsElement>` type, which specifies that the platform-specific can be invoked on `Label` instances from iOS.

## Creating the Effect

The `shadow` platform-specific adds the `MyCompany.LabelShadowEffect` to a `Label`, and removes it. The following code example shows the `LabelShadowEffect` implementation for the iOS project:

```
[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace ShadowPlatformSpecific.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached()
        {
            UpdateShadow();
        }

        protected override void OnDetached()
        {
        }

        protected override void OnElementPropertyChanged(PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == Shadow.IsShadowedProperty.PropertyName)
            {
                UpdateShadow();
            }
        }

        void UpdateShadow()
        {
            try
            {
                if (((Label)Element).OnThisPlatform().IsShadowed())
                {
                    Control.Layer.CornerRadius = 5;
                    Control.Layer.ShadowColor = UIColor.Black.CGColor;
                    Control.Layer.ShadowOffset = new CGSize(5, 5);
                    Control.Layer.ShadowOpacity = 1.0f;
                }
                else if (!((Label)Element).OnThisPlatform().IsShadowed())
                {
                    Control.Layer.ShadowOpacity = 0;
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}
```

The `UpdateShadow` method sets `Control.Layer` properties to create the shadow, provided that the `IsShadowed` attached property is set to `true`, and provided that the `Shadow` platform-specific has been invoked on the same platform that the Effect is implemented for. This check is performed with the `OnThisPlatform` method.

If the `Shadow.IsShadowed` attached property value changes at runtime, the Effect needs to respond by removing

the shadow. Therefore, an overridden version of the `OnElementPropertyChanged` method is used to respond to the bindable property change by calling the `UpdateShadow` method.

For more information about creating an effect, see [Creating an Effect](#) and [Passing Effect Parameters as Attached Properties](#).

## Consuming a Platform-Specific

The `Shadow` platform-specific is consumed in XAML by setting the `Shadow.IsShadowed` attached property to a `boolean` value:

```
<ContentPage xmlns:ios="clr-namespace:MyCompany.Forms.PlatformConfiguration.iOS" ...>
    ...
    <Label Text="Label Shadow Effect" ios:Shadow.IsShadowed="true" ... />
    ...
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using MyCompany.Forms.PlatformConfiguration.iOS;

...
shadowLabel.On<iOS>().SetIsShadowed(true);
```

For more information about consuming platform-specifics, see [Consuming Platform-Specifics](#).

## Summary

This article demonstrated how to expose an Effect through a platform-specific. The result is an Effect that can be more easily consumed through XAML, and through a fluent code API.

## Related Links

- [ShadowPlatformSpecific \(sample\)](#)
- [Customizing Controls with Effects](#)
- [Attached Properties](#)

# Consuming and Creating Xamarin.Forms Plugins

11/11/2018 • 2 minutes to read • [Edit Online](#)

There are many native platform features that exist across all platforms but have slightly different APIs. One way for developers to use these features is by creating an abstract cross-platform interface, and then implementing that interface in the various platforms. The Xamarin.Forms application then accesses these platform implementations using [DependencyService](#).

Developers can share this work by writing a *plugin* and publishing it to NuGet.

## NOTE

Many cross-platform features previously available only through plugins are now part of the open-source [Xamarin.Essentials](#) library. These features include: battery status, compass, motion sensors, geolocation, text-to-speech, and a lot more. In the future, [Xamarin.Essentials](#) will be the primary source of cross-platform features for Xamarin.Forms applications. Although developers can still create and publish plugins, consider contributing to [Xamarin.Essentials](#).

## Finding and Adding Plugins

The Xamarin community has created many cross-platform plugins compatible with Xamarin.Forms. A large collection can be found at:

### Xamarin Plugins

For a guide to adding NuGet packages to your project, see our walkthrough on [including a NuGet package in your project](#).

## Creating plugins

It's also possible to create and publish your own plugins as Nuget packages (and Xamarin Components). Many existing plugins are open-source so you can review their code to understand how they have been written.

For example, the list of plugins below are all open source, and they correspond to some samples in the [DependencyService](#) section:

- **Text-to-Speech** by James Montemagno – [GitHub](#) and [NuGet](#)
- **Battery Status** by James Montemagno – [GitHub](#) and [NuGet](#)

Those Github projects can provide a good starting point for creating your own cross-platform plugins, as do these instructions for [creating a plugin for Xamarin](#).

### Structuring Cross-Platform Plugin Projects

Although there are no particular requirements for designing a NuGet package, there are some guidelines for creating a package for cross-platform apps.

In the past, a cross-platform plugin generally consisted of the following components:

- PCL with an Interface that represents the API for the plugin,
- iOS, Android, and Universal Windows Platform (UWP) class libraries with an implementation of the Interface.

Read James Montemagno's [blog post](#) describing the process of creating plugins for Xamarin.

More recently, plugins can be created with a single multi-targeted platform. This approach is discussed in James

Montemagno's [blog post](#). This approach is used in James Montemagno's plugins linked above, and is also the format used in **Xamarin.Essentials**.

It is a preferable to avoid referencing Xamarin.Forms directly from a plug-in. This can create version-conflict issues when other developers attempt to use the plug-in. Instead try to design the API so that it can be used by any Xamarin or .NET application.

### Publishing NuGet Packages

NuGet packages have a **nuspec** file, which is an xml file that defines which parts of your project are published in the package. The **nuspec** file also includes information about the package, such as id, title, and authors.

See [NuGet's documentation](#) for more information about creating and publishing NuGet packages.

## Related Links

- [Creating Reusable Plugins for Xamarin.Forms](#)
- [Using & Developing Plugins for Xamarin \(video\)](#)

# Tizen .NET

10/3/2018 • 2 minutes to read • [Edit Online](#)

*Tizen .NET allows you to develop Tizen applications to run on Samsung devices, including TVs, wearables, mobile devices, and other IoT devices.*

Tizen .NET enables you to build .NET applications with Xamarin.Forms and the Tizen .NET framework. Xamarin.Forms allows you to easily create user interfaces, while the TizenFX API provides interfaces to the hardware that's found in modern TV, mobile, wearable, and IoT devices. For more information about Tizen .NET, see [Introduction to Tizen .NET Application](#).

## Get started

Before you can start developing Tizen .NET applications, you must first set up your development environment. For more information, see [Installing Visual Studio Tools for Tizen](#).

For information about how to add Tizen .NET project to an existing Xamarin.Forms solution, see [Creating your First Tizen .NET Application](#).

## Documentation

- [Xamarin.Forms documentation](#) – how to build cross-platform applications with C# and Xamarin.Forms.
- [developer.tizen.org](#) – documentation and videos to help you build and deploy Tizen applications.

## Samples

Samsung maintains a fork of the [Xamarin.Forms samples with Tizen projects added](#), and there is a separate repository [Tizen-Csharp-Samples](#) that contains additional projects, including Wearable and TV-specific demos.

# Windows Platform Features

6/8/2018 • 2 minutes to read • [Edit Online](#)

Developing Xamarin.Forms applications for Windows platforms requires Visual Studio. The [requirements page](#) contains more information about the pre-requisites.



## Platform Support

The Xamarin.Forms templates available in Visual Studio contain a Universal Windows Platform (UWP) project.

### NOTE

Xamarin.Forms 1.x and 2.x support *Windows Phone 8 Silverlight*, *Windows Phone 8.1*, and *Windows 8.1* application development. However, these project types have been deprecated.

## Getting Started

Go to **File > New > Project** in Visual Studio and choose one of the **Cross-Platform > Blank App (Xamarin.Forms)** templates to get started.

Older Xamarin.Forms solutions, or those created on macOS, will not have all the Windows projects listed above (but they need to be manually added). If the Windows platform you wish to target isn't already in your solution, visit the [setup instructions](#) to add the desired Windows project type/s.

## Samples

All the [samples](#) for Charles Petzold's book *Creating Mobile Apps with Xamarin.Forms* include Universal Windows Platform (for Windows 10) projects.

The "[Scott Hanselman](#)" demo app is available separately, and also includes Apple Watch and Android Wear projects (using Xamarin.iOS and Xamarin.Android respectively, Xamarin.Forms does not run on those platforms).

## Related Links

- [Setup Windows Projects](#)

# Setup Windows Projects

10/12/2018 • 3 minutes to read • [Edit Online](#)

*Adding new Windows projects to an existing Xamarin.Forms solution*

Older Xamarin.Forms solutions (or those created on macOS) will not have Universal Windows Platform (UWP) app projects. Therefore, you'll need to manually add a UWP project to build a Windows 10 (UWP) app.

## Add a Universal Windows Platform app

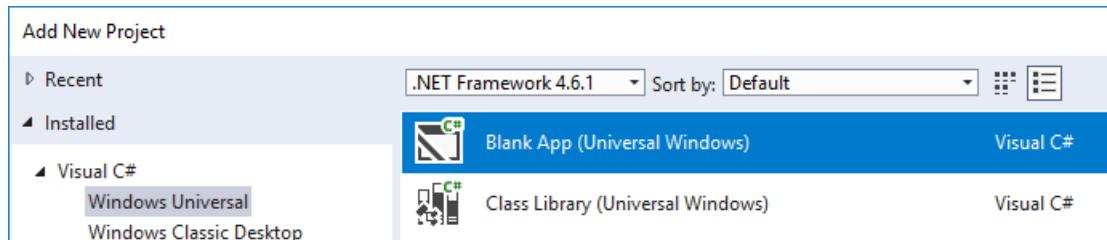
You should be running **Visual Studio 2017** on **Windows 10** to build UWP apps. For more information about the Universal Windows Platform, see [Intro to the Universal Windows Platform](#).

UWP is available in Xamarin.Forms 2.1 and later, and Xamarin.Forms.Maps is supported in Xamarin.Forms 2.2 and later.

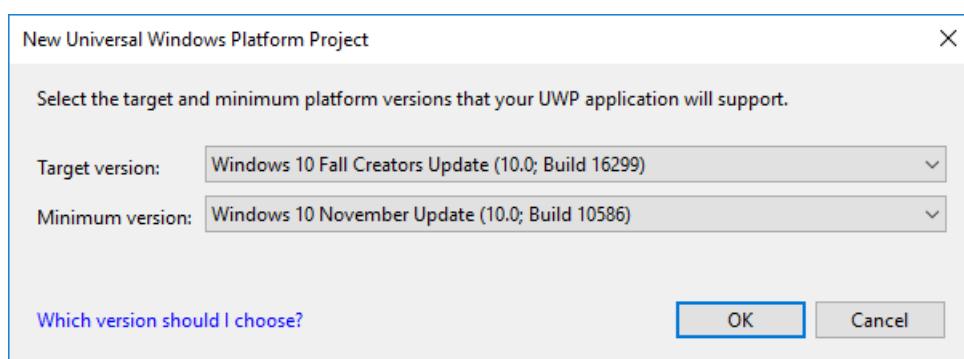
Check the [troubleshooting](#) section for helpful tips.

Follow these instructions to add a UWP app that will run on Windows 10 phones, tablets, and desktops:

- 1 . Right-click on the solution and select **Add > New Project...** and add a **Blank App (Universal Windows)** project:



- 2 . In the **New Universal Windows Platform Project** dialog, select the minimum and target versions of Windows 10 that the app will run on:

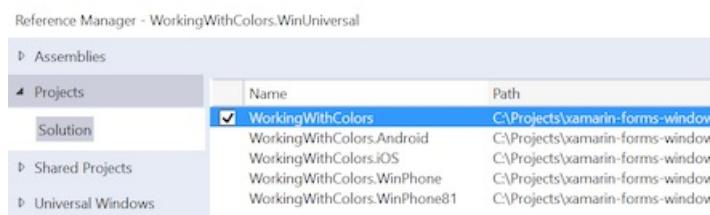


- 3 . Right-click on the UWP project and select **Manage NuGet Packages...** and add the **Xamarin.Forms** package. Ensure the other projects in the solution are also updated to the same version of the Xamarin.Forms package.

- 4 . Make sure the new UWP project will be built in the **Build > Configuration Manager** window (this probably won't have happened by default). Tick the **Build** and **Deploy** boxes for the Universal project:

Active solution configuration:		Active solution platform:		
Debug	▼	Any CPU	▼	▼
Project	Configuration	Platform	Build	Deploy
BugSweeper	Debug	Any CPU	<input checked="" type="checkbox"/>	<input type="checkbox"/>
BugSweeper.Android	Debug	Any CPU	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
BugSweeper.iOS	Debug	iPhone	<input type="checkbox"/>	<input type="checkbox"/>
BugSweeper.WinApp	Debug	Any CPU	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
BugSweeper.WinPhone	Debug	Any CPU	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
BugSweeper.WinPhone81	Debug	Any CPU	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
BugSweeper.WinUniversal	Debug	x86	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

5 . Right-click on the project and select **Add > Reference** and create a reference to the Xamarin.Forms application project (.NET Standard or Shared Project).



6 . In the UWP project, edit **App.xaml.cs** to include the `Init` method call inside the `OnLaunched` method around line 52:

```
// under this line
rootFrame.NavigationFailed += OnNavigationFailed;
// add this line
Xamarin.Forms.Forms.Init (e); // requires the `e` parameter
```

7 . In the UWP project, edit **MainPage.xaml** by removing the `Grid` contained within the `Page` element.

8 . In **MainPage.xaml**, add a new `xmlns` entry for `Xamarin.Forms.Platform.UWP`:

```
xmlns:forms="using:Xamarin.Forms.Platform.UWP"
```

9 . In **MainPage.xaml**, change the root `<Page>` element to `<forms:WindowsPage>`:

```
<forms:WindowsPage
...
  xmlns:forms="using:Xamarin.Forms.Platform.UWP"
...
</forms:WindowsPage>
```

10 . In the UWP project, edit **MainPage.xaml.cs** to remove the `: Page` inheritance specifier for the class name (since it will now inherit from `WindowsPage` due to the change made in the previous step):

```
public sealed partial class MainPage // REMOVE ": Page"
```

11 . In **MainPage.xaml.cs**, add the `LoadApplication` call in the `MainPage` constructor to start the Xamarin.Forms app:

```
// below this existing line  
this.InitializeComponent();  
// add this line  
LoadApplication(new YOUR_NAMESPACE.App());
```

12 . Add any local resources (eg. image files) from the existing platform projects that are required.

## Troubleshooting

### "Target Invocation Exception" when using "Compile with .NET Native tool chain"

If your UWP app is referencing multiple assemblies (for example third party control libraries, or your app itself is split into multiple libraries), Xamarin.Forms may be unable to load objects from those assemblies (such as custom renderers).

This might occur when using the **Compile with .NET Native tool chain** which is an option for UWP apps in the **Properties > Build > General** window for the project.

You can fix this by using a UWP-specific overload of the `Forms.Init` call in `App.xaml.cs` as shown in the code below (you should replace `ClassInOtherAssembly` with an actual class your code references):

```
// You'll need to add `using System.Reflection;`  
List<Assembly> assembliesToInclude = new List<Assembly>();  
  
// Now, add in all the assemblies your app uses  
assembliesToInclude.Add(typeof (ClassInOtherAssembly).GetTypeInfo().Assembly);  
  
// Also do this for all your other 3rd party libraries  
Xamarin.Forms.Forms.Init(e, assembliesToInclude);  
// replaces Xamarin.Forms.Forms.Init(e);
```

Add an entry for each assembly that you have added as a reference in the Solution Explorer, either via a direct reference or a NuGet.

### Dependency Services and .NET Native Compilation

Release builds using .NET Native compilation can fail to resolve dependency services that are defined outside the main app executable (such as in a separate project or library).

Use the `DependencyService.Register<T>()` method to manually register dependency service classes. Based on the example above, add the register method like this:

```
Xamarin.Forms.Forms.Init(e, assembliesToInclude);  
Xamarin.Forms.DependencyService.Register<ClassInOtherAssembly>(); // add this
```

# WPF Platform Setup

11/12/2018 • 3 minutes to read • [Edit Online](#)



Xamarin.Forms now has preview support for the Windows Presentation Foundation (WPF). This article demonstrates how to add a WPF project to a Xamarin.Forms solution.

Before you start, create a new Xamarin.Forms solution in Visual Studio 2017, or use an existing Xamarin.Forms solution, for example, [BoxViewClock](#). You can only add WPF apps to a Xamarin.Forms solution in Windows.

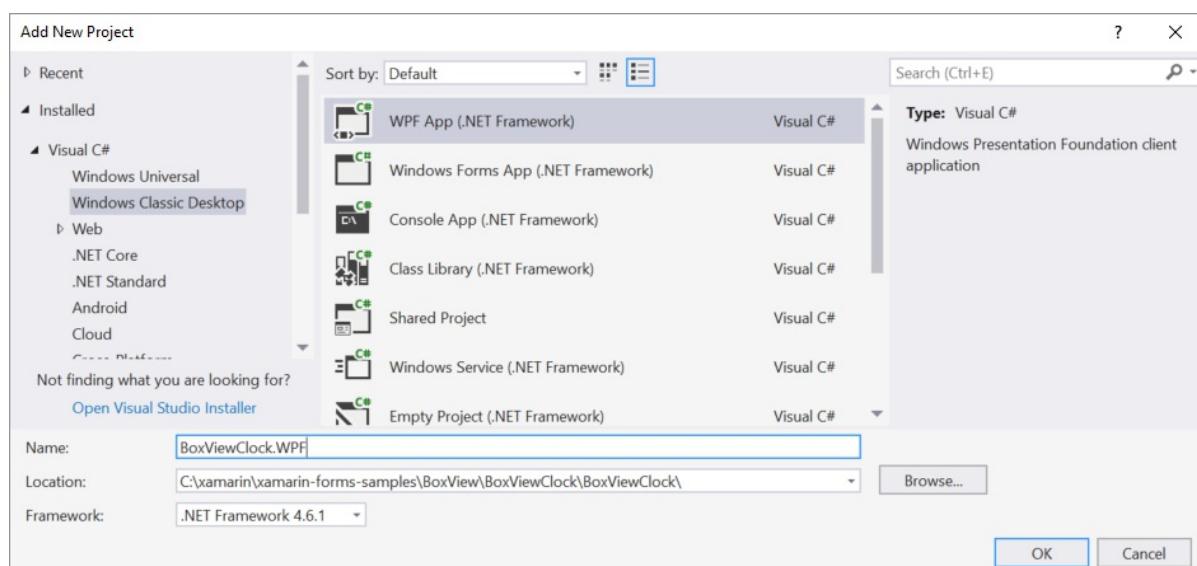
## Add a WPF project to a Xamarin.Forms app with Xamarin.University

### [Xamarin.Forms 3.0 WPF Support, by Xamarin University](#)

## Adding a WPF App

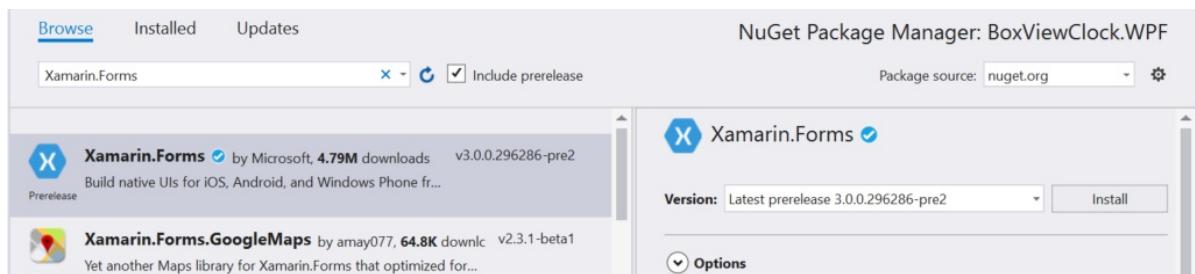
Follow these instructions to add a WPF app that will run on the Windows 7, 8, and 10 desktops:

1. In Visual Studio 2017, right-click on the solution name in the **Solution Explorer** and choose **Add > New Project....**
2. In the **New Project** window, at the left select **Visual C#** and **Windows Classic Desktop**. In the list of project types, choose **WPF App (.NET Framework)**.
3. Type a name for the project with a **WPF** extension, for example, **BoxViewClock.WPF**. Click the **Browse** button, select the **BoxViewClock** folder, and press **Select Folder**. This will put the WPF project in the same directory as the other projects in the solution.



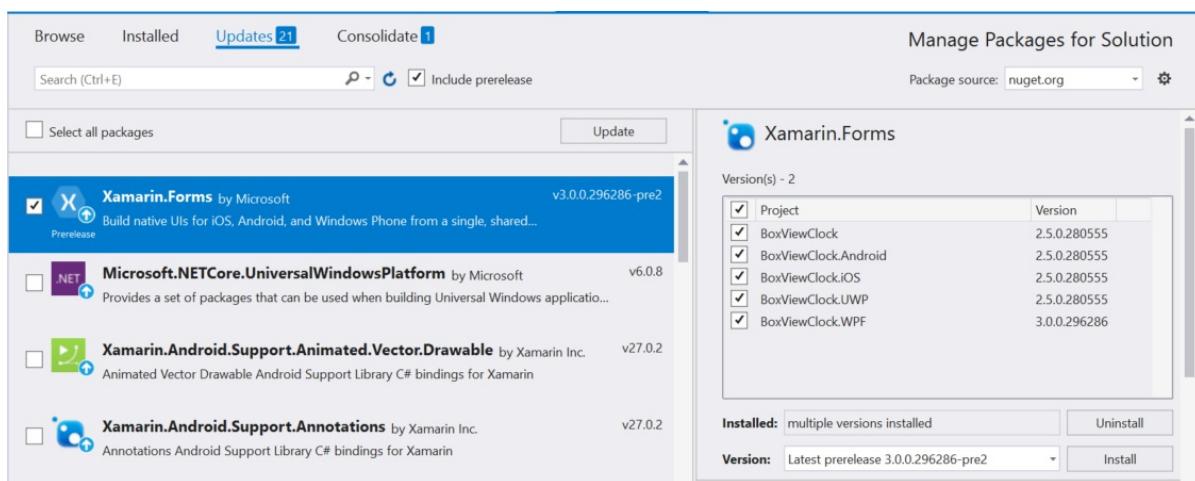
Press OK to create the project.

4. In the **Solution Explorer**, right click the new **BoxViewClock.WPF** project and select **Manage NuGet Packages**. Select the **Browse** tab, click the **Include prerelease** checkbox, and search for **Xamarin.Forms**.

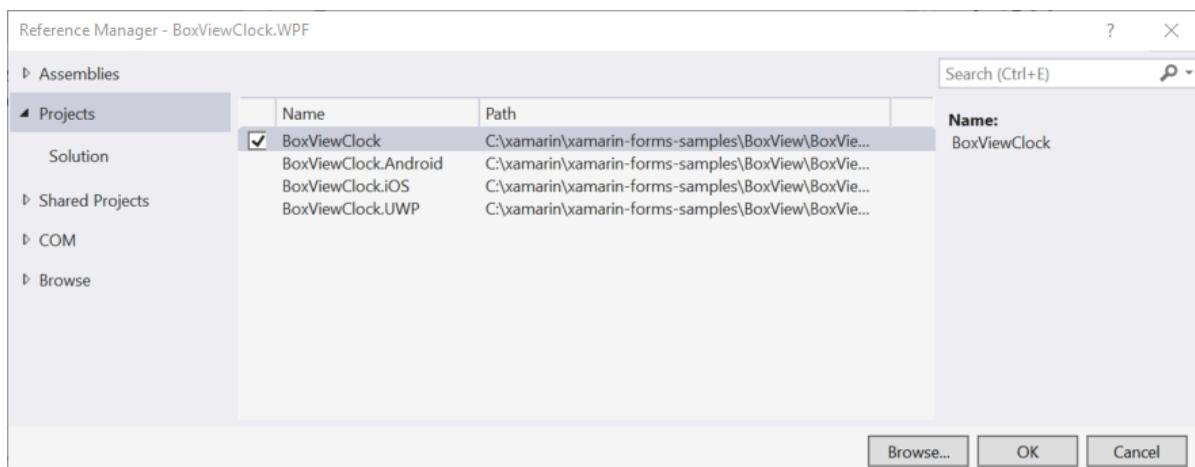


Select that package and click the **Install** button.

5. Now search for **Xamarin.Forms.Platform.WPF** package and install that one as well. Make sure the package is from Microsoft!
6. Right click the solution name in the **Solution Explorer** and select **Manage NuGet Packages for Solution**. Select the **Update** tab and the **Xamarin.Forms** package. Select all the projects and update them to the same Xamarin.Forms version:



7. In the WPF project, right-click on **References**. In the **Reference Manager** dialog, select **Projects** at the left, and check the checkbox adjacent to the **BoxViewClock** project:



8. Edit the **MainWindow.xaml** file of the WPF project. In the `Window` tag, add an XML namespace declaration for the **Xamarin.Forms.Platform.WPF** assembly and namespace:

```
xmlns:wpf="clr-namespace:Xamarin.Forms.Platform.WPF;assembly=Xamarin.Forms.Platform.WPF"
```

Now change the `Window` tag to `wpf:FormsApplicationPage`. Change the `Title` setting to the name of your application, for example, **BoxViewClock**. The completed XAML file should look like this:

```

<wpf:FormsApplicationPage x:Class="BoxViewClock.WPF.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:wpf="clr-namespace:Xamarin.Forms.Platform.WPF;assembly=Xamarin.Forms.Platform.WPF"
    xmlns:local="clr-namespace:BoxViewClock.WPF"
    mc:Ignorable="d"
    Title="BoxViewClock" Height="450" Width="800">
<Grid>
</Grid>
</wpf:FormsApplicationPage>

```

9. Edit the **MainWindow.xaml.cs** file of the WPF project. Add two new `using` directives:

```

using Xamarin.Forms;
using Xamarin.Forms.Platform.WPF;

```

Change the base class of `MainWindow` from `Window` to `FormsApplicationPage`. Following the `InitializeComponent` call, add the following two statements:

```

Forms.Init();
LoadApplication(new BoxViewClock.App());

```

Except for comments and unused `using` directives, the complete **MainWindows.xaml.cs** file should look like this:

```

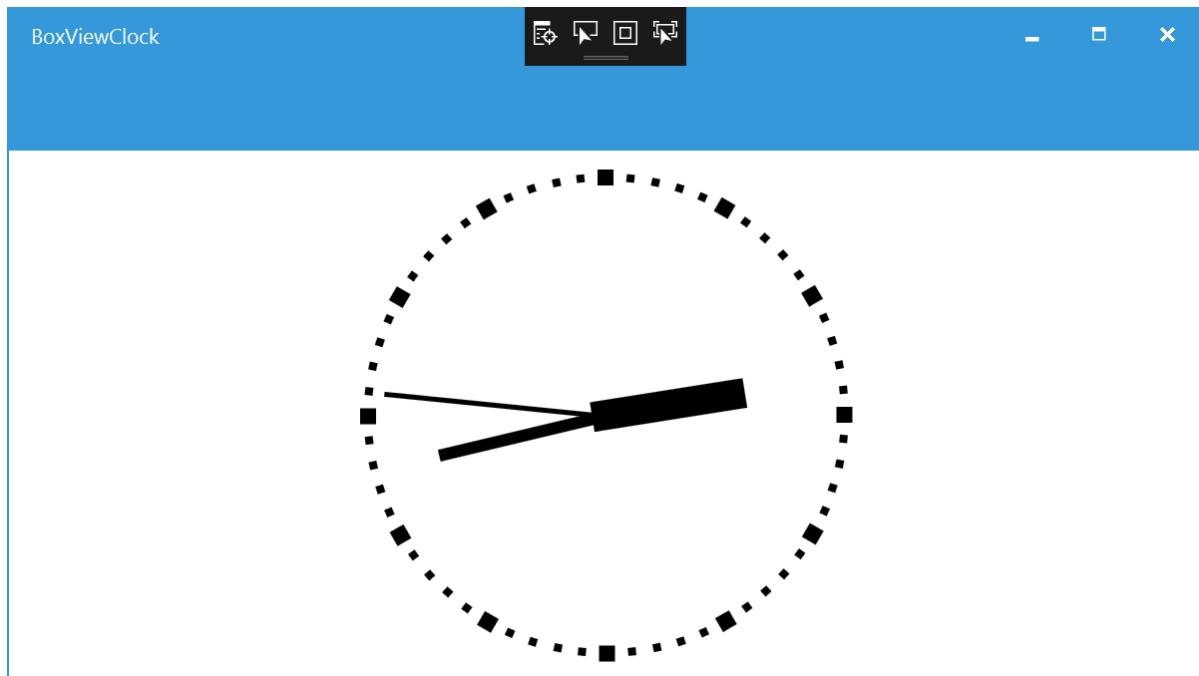
using Xamarin.Forms;
using Xamarin.Forms.Platform.WPF;

namespace BoxViewClock.WPF
{
    public partial class MainWindow : FormsApplicationPage
    {
        public MainWindow()
        {
            InitializeComponent();

            Forms.Init();
            LoadApplication(new BoxViewClock.App());
        }
    }
}

```

10. Right-click the WPF project in the **Solution Explorer** and select **Set as Startup Project**. Press F5 to run the program with the Visual Studio debugger on the Windows desktop:



## Next Steps

### Platform Specifics

You can determine what platform your Xamarin.Forms application is running on from either code or XAML. This allows you to change program characteristics when it's running on WPF. In code, compare the value of `Device.RuntimePlatform` with the `Device.WPF` constant (which equals the string "WPF"). If there's a match, the application is running on WPF.

In XAML, you can use the `OnPlatform` tag to select a property value specific to the platform:

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White" />
        <On Platform="macOS" Value="White" />
        <On Platform="Android" Value="Black" />
        <On Platform="WPF" Value="Blue" />
    </OnPlatform>
</Button.TextColor>
```

### Window Size

You can adjust the initial size of the window in the WPF `MainWindow.xaml` file:

```
Title="BoxViewClock" Height="450" Width="800"
```

## Issues

This is a Preview, so you should expect that not everything is production ready. Not all NuGet packages for Xamarin.Forms are ready for WPF, and some features might not be fully working.

# Xamarin.Essentials

10/9/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

Xamarin.Essentials provides developers with cross-platform APIs for their mobile applications.

Android, iOS, and UWP offer unique operating system and platform APIs that developers have access to all in C# leveraging Xamarin. Xamarin.Essentials provides a single cross-platform API that works with any Xamarin.Forms, Android, iOS, or UWP application that can be accessed from shared code no matter how the user interface is created.

## Get Started with Xamarin.Essentials

Follow the [getting started guide](#) to install the **Xamarin.Essentials** NuGet package into your existing or new Xamarin.Forms, Android, iOS, or UWP projects.

## Feature Guides

Follow the guides to integrate these Xamarin.Essentials features into your applications:

- [Accelerometer](#) – Retrieve acceleration data of the device in three dimensional space.
- [App Information](#) – Find out information about the application.
- [Barometer](#) – Monitor the barometer for pressure changes.
- [Battery](#) – Easily detect battery level, source, and state.
- [Clipboard](#) – Quickly and easily set or read text on the clipboard.
- [Compass](#) – Monitor compass for changes.
- [Connectivity](#) – Check connectivity state and detect changes.
- [Data Transfer](#) – Send text and website uris to other apps.
- [Device Display Information](#) – Get the device's screen metrics and orientation.
- [Device Information](#) – Find out about the device with ease.
- [Email](#) – Easily send email messages.
- [File System Helpers](#) – Easily save files to app data.
- [Flashlight](#) – A simple way to turn the flashlight on/off.
- [Geocoding](#) – Geocode and reverse geocode addresses and coordinates.
- [Geolocation](#) – Retrieve the device's GPS location.
- [Gyroscope](#) – Track rotation around the device's three primary axes.
- [Launcher](#) – Enables an application to open a URI by the system.
- [Magnetometer](#) – Detect device's orientation relative to Earth's magnetic field.
- [MainThread](#) – Run code on the application's main thread.
- [Maps](#) – Open the maps application to a specific location.
- [Open Browser](#) – Quickly and easily open a browser to a specific website.
- [Orientation Sensor](#) – Retrieve the orientation of the device in three dimensional space.
- [Phone Dialer](#) – Open the phone dialer.

- [Power](#) – Obtain the device's energy-saver status.
- [Preferences](#) – Quickly and easily add persistent preferences.
- [Screen Lock](#) – Keep the device screen awake.
- [Secure Storage](#) – Securely store data.
- [SMS](#) – Create an SMS message for sending.
- [Text-to-Speech](#) – Vocalize text on the device.
- [Version Tracking](#) – Track the applications version and build numbers.
- [Vibrate](#) – Make the device vibrate.

## Troubleshooting

Find help if you are running into issues.

## API Documentation

Browse the API documentation for every feature of Xamarin.Essentials.

# Get Started with Xamarin.Essentials

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

Xamarin.Essentials provides a single cross-platform API that works with any iOS, Android, or UWP application that can be accessed from shared code no matter how the user interface is created.

## Platform Support

Xamarin.Essentials supports the following platforms and operating systems:

PLATFORM	VERSION
Android	4.4 (API 19) or higher
iOS	10.0 or higher
UWP	10.0.16299.0 or higher

## Installation

Xamarin.Essentials is available as a NuGet package that can be added to any existing or new project using Visual Studio.

1. Download and install [Visual Studio](#) with the [Visual Studio tools for Xamarin](#).
2. Open an existing project, or create a new project using the Blank App template under **Visual Studio C#** (Android, iPhone & iPad, or Cross-Platform). **Important:** If adding to a UWP project ensure Build 16299 or higher is set in the project properties.
3. Add the **Xamarin.Essentials** NuGet package to each project:
  - [Visual Studio](#)
  - [Visual Studio for Mac](#)

In the Solution Explorer panel, right click on the solution name and select **Manage NuGet Packages**. Search for **Xamarin.Essentials** and install the package into **ALL** projects including Android, iOS, UWP, and .NET Standard libraries.

### TIP

Check the **Include prerelease** box while the **Xamarin.Essentials** NuGet is in preview.

4. Add a reference to Xamarin.Essentials in any C# class to reference the APIs.

```
using Xamarin.Essentials;
```

## 5. Xamarin.Essentials requires platform-specific setup:

- [Android](#)
- [iOS](#)
- [UWP](#)

Xamarin.Essentials supports a minimum Android version of 4.4, corresponding to API level 19, but the target Android version for compiling must be 8.1, corresponding to API level 27. (In Visual Studio, these two versions are set in the Project Properties dialog for the Android project, in the Android Manifest tab. In Visual Studio for Mac, they're set in the Project Options dialog for the Android project, in the Android Application tab.)

Xamarin.Essentials installs version 27.0.2.1 of the `Xamarin.Android.Support` libraries that it requires. Any other `Xamarin.Android.Support` libraries that your application requires should also be updated to version 27.0.2.1 using the NuGet package manager. All `Xamarin.Android.Support` libraries used by your application should be the same, and should be at least version 27.0.2.1. Refer to the [troubleshooting page](#) if you have issues adding the `Xamarin.Essentials` NuGet or updating NuGets in your solution.

In the Android project's `MainLauncher` or any `Activity` that is launched `Xamarin.Essentials` must be initialized in the `OnCreate` method:

```
protected override void OnCreate(Bundle savedInstanceState) {  
    //...  
    base.OnCreate(savedInstanceState);  
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code  
    //...
```

To handle runtime permissions on Android, `Xamarin.Essentials` must receive any `OnRequestPermissionsResult`. Add the following code to all `Activity` classes:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,  
[GeneratedEnum] Android.Content.PM.Permission[] grantResults)  
{  
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);  
  
    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);  
}
```

## 6. Follow the [Xamarin.Essentials guides](#) that enable you to copy and paste code snippets for each feature.

## Xamarin.Essentials - Cross-Platform APIs for Mobile Apps (video)

## Other Resources

We recommend developers new to Xamarin visit [getting started with Xamarin development](#).

Visit the [Xamarin.Essentials GitHub Repository](#) to see the current source code, what is coming next, run samples, and clone the repository. Community contributions are welcome!

Browse through the [API documentation](#) for every feature of `Xamarin.Essentials`.

# Xamarin.Essentials: Accelerometer

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Accelerometer** class lets you monitor the device's accelerometer sensor, which indicates the acceleration of the device in three-dimensional space.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Accelerometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Accelerometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the acceleration. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```

public class AccelerometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public AccelerometerTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Accelerometer.RadingChanged += Accelerometer_ReadingChanged;
    }

    void Accelerometer_ReadingChanged(object sender, AccelerometerChangedEventArgs e)
    {
        var data = e.Rading;
        Console.WriteLine($"Reading: X: {data.Acceleration.X}, Y: {data.Acceleration.Y}, Z: {data.Acceleration.Z}");
        // Process Acceleration X, Y, and Z
    }

    public void ToggleAccelerometer()
    {
        try
        {
            if (Accelerometer.IsMonitoring)
                Accelerometer.Stop();
            else
                Accelerometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}

```

Accelerometer readings are reported back in G. A G is a unit of gravitation force equal to that exerted by the earth's gravitational field ( $9.81 \text{ m/s}^2$ ).

The coordinate-system is defined relative to the screen of the phone in its default orientation. The axes are not swapped when the device's screen orientation changes.

The X axis is horizontal and points to the right, the Y axis is vertical and points up and the Z axis points towards the outside of the front face of the screen. In this system, coordinates behind the screen have negative Z values.

Examples:

- When the device lies flat on a table and is pushed on its left side toward the right, the x acceleration value is positive.
- When the device lies flat on a table, the acceleration value is  $+1.00 \text{ G}$  or  $(+9.81 \text{ m/s}^2)$ , which correspond to the acceleration of the device ( $0 \text{ m/s}^2$ ) minus the force of gravity ( $-9.81 \text{ m/s}^2$ ) and normalized as in G.
- When the device lies flat on a table and is pushed toward the sky with an acceleration of  $A \text{ m/s}^2$ , the acceleration value is equal to  $A+9.81$  which corresponds to the acceleration of the device  $(+A \text{ m/s}^2)$  minus the force of gravity ( $-9.81 \text{ m/s}^2$ ) and normalized in G.

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

## API

- [Accelerometer source code](#)
- [Accelerometer API documentation](#)

# Xamarin.Essentials: App Information

11/15/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **AppInfo** class provides information about your application.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using AppInfo

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

## Obtaining Application Information:

The following information is exposed through the API:

```
// Application Name  
var appName = AppInfo.Name;  
  
// Package Name/Application Identifier (com.microsoft.testapp)  
var packageName = AppInfo.PackageName;  
  
// Application Version (1.0.0)  
var version = AppInfo.VersionString;  
  
// Application Build Number (1)  
var build = AppInfo.BuildString;
```

## Displaying Application Settings

The **AppInfo** class can also display a page of settings maintained by the operating system for the application:

```
// Display settings page  
AppInfo.OpenSettings();
```

This settings page allows the user to change application permissions and perform other platform-specific tasks.

## API

- [AppInfo source code](#)
- [AppInfo API documentation](#)



# Xamarin.Essentials: Barometer

10/31/2018 • 2 minutes to read • [Edit Online](#)



The **Barometer** class lets you monitor the device's barometer sensor, which measures pressure.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Barometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Barometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the barometer's pressure reading in kilopascals. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```

public class BarometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public BarometerTest()
    {
        // Register for reading changes.
        Barometer.RadingChanged += Barometer_ReadingChanged;
    }

    void Barometer_ReadingChanged(object sender, BarometerChangedEventArgs e)
    {
        var data = e.Rading;
        // Process Pressure
        Console.WriteLine($"Reading: Pressure: {data.Pressure} kilopascals");
    }

    public void ToggleBarometer()
    {
        try
        {
            if (Barometer.IsMonitoring)
                Barometer.Stop();
            else
                Barometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}

```

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

No platform-specific implementation details.

## API

- [Barometer source code](#)

- [Barometer API documentation](#)

# Xamarin.Essentials: Battery

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Battery** class lets you check the device's battery information and monitor for changes.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Battery** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The `Battery` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.BatteryStats)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.BATTERY_STATS" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **Battery** permission. This will automatically update the **AndroidManifest.xml** file.

## Using Battery

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Check current battery information:

```

var level = Battery.ChargeLevel; // returns 0.0 to 1.0 or -1.0 if unable to determine.

var state = Battery.State;

switch (state)
{
    case BatteryState.Charging:
        // Currently charging
        break;
    case BatteryState.Full:
        // Battery is full
        break;
    case BatteryState.Discharging:
    case BatteryState.NotCharging:
        // Currently discharging battery or not being charged
        break;
    case BatteryState.NotPresent:
        // Battery doesn't exist in device (desktop computer)
    case BatteryState.Unknown:
        // Unable to detect battery state
        break;
}

var source = Battery.PowerSource;

switch (source)
{
    case BatteryPowerSource.Battery:
        // Being powered by the battery
        break;
    case BatteryPowerSource.AC:
        // Being powered by A/C unit
        break;
    case BatteryPowerSource.Usb:
        // Being powered by USB cable
        break;
    case BatteryPowerSource.Wireless:
        // Powered via wireless charging
        break;
    case BatteryPowerSource.Unknown:
        // Unable to detect power source
        break;
}

```

Whenever any of the battery's properties change an event is triggered:

```

public class BatteryTest
{
    public BatteryTest()
    {
        // Register for battery changes, be sure to unsubscribe when needed
        Battery.BatteryChanged += Battery_BatteryChanged;
    }

    void Battery_BatteryChanged(object sender, BatteryChangedEventArgs e)
    {
        var level = e.ChargeLevel;
        var state = e.State;
        var source = e.PowerSource;
        Console.WriteLine($"Reading: Level: {level}, State: {state}, Source: {source}");
    }
}

```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No platform differences.

## API

- [Battery source code](#)
- [Battery API documentation](#)

# Xamarin.Essentials: Clipboard

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Clipboard** class lets you copy and paste text to the system clipboard between applications.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Clipboard

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To check if the **Clipboard** has text currently ready to be pasted:

```
var hasText = Clipboard.HasText;
```

To set text to the **Clipboard**:

```
Clipboard.SetText("Hello World");
```

To read text from the **Clipboard**:

```
var text = await Clipboard.GetTextAsync();
```

## API

- [Clipboard source code](#)
- [Clipboard API documentation](#)

# Xamarin.Essentials: Compass

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Compass** class lets you monitor the device's magnetic north heading.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Compass

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Compass functionality works by calling the `Start` and `Stop` methods to listen for changes to the compass.

Any changes are sent back through the `ReadingChanged` event. Here is an example:

```

public class CompassTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public CompassTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Compass.RadingChanged += Compass_RadingChanged;
    }

    void Compass_RadingChanged(object sender, CompassChangedEventArgs e)
    {
        var data = e.Rading;
        Console.WriteLine($"Reading: {data.HeadingMagneticNorth} degrees");
        // Process Heading Magnetic North
    }

    public void ToggleCompass()
    {
        try
        {
            if (Compass.IsMonitoring)
                Compass.Stop();
            else
                Compass.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Some other exception has occurred
        }
    }
}

```

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

## Platform Implementation Specifics

- [Android](#)

Android does not provide a API for retrieving the compass heading. We utilize the accelerometer and magnetometer to calculate the magnetic north heading, which is recommended by Google.

In rare instances, you may see inconsistent results because the sensors need to be calibrated, which involves moving your device in a figure-8 motion. The best way of doing this is to open Google Maps, tap on the dot for your location, and select **Calibrate compass**.

Be aware that running multiple sensors from your app at the same time may adjust the sensor speed.

## Low Pass Filter

Due to how the Android compass values are updated and calculated there may be a need to smooth out the values. A *Low Pass Filter* can be applied that averages the sine and cosine values of the angles and can be turned on by setting the `ApplyLowPassFilter` property on the `Compass` class:

```
Compass.ApplyLowPassFilter = true;
```

This is only applied on the Android platform. More information can be read [here](#).

## API

- [Compass source code](#)
- [Compass API documentation](#)

# Xamarin.Essentials: Connectivity

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Connectivity** class lets you monitor for changes in the device's network conditions, check the current network access, and how it is currently connected.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Connectivity** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The `AccessNetworkState` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessNetworkState)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **Access Network State** permission. This will automatically update the **AndroidManifest.xml** file.

## Using Connectivity

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Check current network access:

```

var current = Connectivity.NetworkAccess;

if (current == NetworkAccess.Internet)
{
    // Connection to internet is available
}

```

[Network access](#) falls into the following categories:

- **Internet** – Local and internet access.
- **ConstrainedInternet** – Limited internet access. Indicates captive portal connectivity, where local access to a web portal is provided, but access to the Internet requires that specific credentials are provided via a portal.
- **Local** – Local network access only.
- **None** – No connectivity is available.
- **Unknown** – Unable to determine internet connectivity.

You can check what type of [connection profile](#) the device is actively using:

```

var profiles = Connectivity.Profiles;
if (profiles.Contains(ConnectionProfile.WiFi))
{
    // Active Wi-Fi connection.
}

```

Whenever the connection profile or network access changes you can receive an event when triggered:

```

public class ConnectivityTest
{
    public ConnectivityTest()
    {
        // Register for connectivity changes, be sure to unsubscribe when finished
        Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;
    }

    void Connectivity_ConnectivityChanged(object sender, ConnectivityChangedEventArgs e)
    {
        var access = e.NetworkAccess;
        var profiles = e.Profiles;
    }
}

```

## Limitations

It is important to note that it is possible that `Internet` is reported by `NetworkAccess` but full access to the web is not available. Due to how connectivity works on each platform it can only guarantee that a connection is available. For instance the device may be connected to a Wi-Fi network, but the router is disconnected from the internet. In this instance Internet may be reported, but an active connection is not available.

## API

- [Connectivity source code](#)
- [Connectivity API documentation](#)

# Xamarin.Essentials: Data Transfer

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **DataTransfer** class enables an application to share data such as text and web links to other applications on the device.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Data Transfer

Add a reference to Xamarin.Essentials in your class:

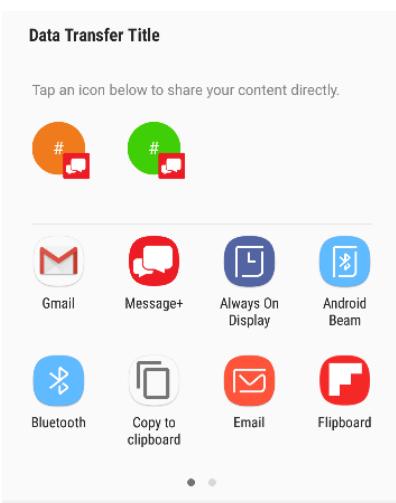
```
using Xamarin.Essentials;
```

The Data Transfer functionality works by calling the `RequestAsync` method with a data request payload that includes information to share to other applications. Text and Uri can be mixed and each platform will handle filtering based on content.

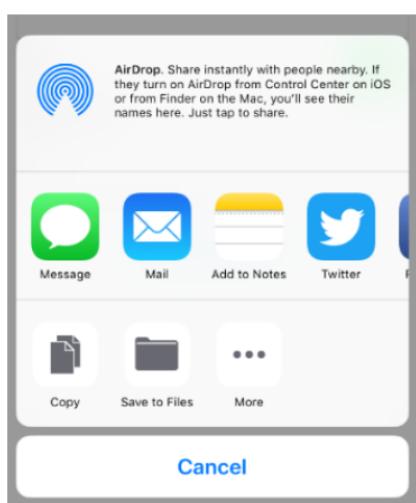
```
public class DataTransferTest
{
    public async Task ShareText(string text)
    {
        await DataTransfer.RequestAsync(new ShareTextRequest
        {
            Text = text,
            Title = "Share Text"
        });
    }

    public async Task ShareUri(string uri)
    {
        await DataTransfer.RequestAsync(new ShareTextRequest
        {
            Uri = uri,
            Title = "Share Web Link"
        });
    }
}
```

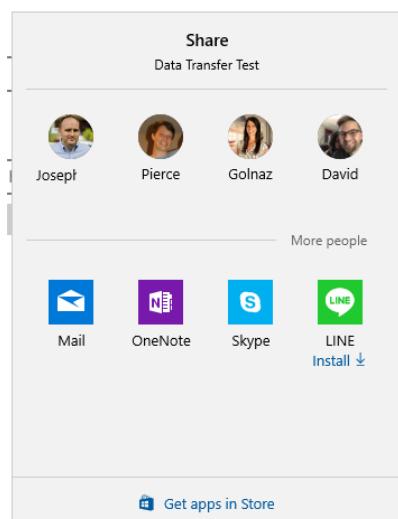
User interface to share to external application that appears when request is made:



Android



iOS



UWP

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- `Subject` property is used for desired subject of a message.

## API

- [Data Transfer source code](#)
- [Data Transfer API documentation](#)

# Xamarin.Essentials: Device Display Information

10/31/2018 • 2 minutes to read • [Edit Online](#)



The **DeviceDisplay** class provides information about the device's screen metrics the application is running on.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using DeviceDisplay

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

## Screen Metrics

In addition to basic device information the **DeviceDisplay** class contains information about the device's screen and orientation.

```
// Get Metrics
var metrics = DeviceDisplay.ScreenMetrics;

// Orientation (Landscape, Portrait, Square, Unknown)
var orientation = metrics.Orientation;

// Rotation (0, 90, 180, 270)
var rotation = metrics.Rotation;

// Width (in pixels)
var width = metrics.Width;

// Height (in pixels)
var height = metrics.Height;

// Screen density
var density = metrics.Density;
```

The **DeviceDisplay** class also exposes an event that can be subscribed to that is triggered whenever any screen metric changes:

```
public class ScreenMetricsTest
{
    public ScreenMetricsTest()
    {
        // Subscribe to changes of screen metrics
        DeviceDisplay.ScreenMetricsChanged += OnScreenMetricsChanged;
    }

    void OnScreenMetricsChanged(ScreenMetricsChangedEventArgs e)
    {
        // Process changes
        var metrics = e.Metrics;
    }
}
```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No differences.

## API

- [DeviceDisplay source code](#)
- [DeviceDisplay API documentation](#)

# Xamarin.Essentials: Device Information

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **DeviceInfo** class provides information about the device the application is running on.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using DeviceInfo

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The following information is exposed through the API:

```
// Device Model (SMG-950U, iPhone10,6)
var device = DeviceInfo.Model;

// Manufacturer (Samsung)
var manufacturer = DeviceInfo.Manufacturer;

// Device Name (Motz's iPhone)
var deviceName = DeviceInfo.Name;

// Operating System Version Number (7.0)
var version = DeviceInfo.VersionString;

// Platform (Android)
var platform = DeviceInfo.Platform;

// Idiom (Phone)
var idiom = DeviceInfo.Idiom;

// Device Type (Physical)
var deviceType = DeviceInfo.DeviceType;
```

## Platforms

`DeviceInfo.Platform` correlates to a constant string that maps to the operating system. The values can be checked with the `Platforms` class:

- **DeviceInfo.Platforms.iOS** – iOS
- **DeviceInfo.Platforms.Android** – Android
- **DeviceInfo.Platforms.UWP** – UWP
- **DeviceInfo.Platforms.Unsupported** – Unsupported

## Idioms

`DeviceInfo.Idiom` correlates a constant string that maps to the type of device the application is running on. The values can be checked with the `Idioms` class:

- `DeviceInfo.Idioms.Phone` – Phone
- `DeviceInfo.Idioms.Tablet` – Tablet
- `DeviceInfo.Idioms.Desktop` – Desktop
- `DeviceInfo.Idioms.TV` – TV
- `DeviceInfo.Idioms.Unsupported` – Unsupported

## Device Type

`DeviceInfo.DeviceType` correlates an enumeration to determine if the application is running on a physical or virtual device. A virtual device is a simulator or emulator.

## Platform Implementation Specifics

- [iOS](#)

iOS does not expose an API for developers to get the name of the specific iOS device. Instead a hardware identifier is returned such as *iPhone10,6* which refers to the iPhone X. A mapping of these identifiers are not provided by Apple, but can be found on [The iPhone Wiki](#) (a non-official source source).

## API

- [DeviceInfo source code](#)
- [DeviceInfo API documentation](#)

# Xamarin.Essentials: Email

11/15/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Email** class enables an application to open the default email application with a specified information including subject, body, and recipients (TO, CC, BCC).

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Email

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Email functionality works by calling the `ComposeAsync` method an `EmailMessage` that contains information about the email:

```
public class EmailTest
{
    public async Task SendEmail(string subject, string body, List<string> recipients)
    {
        try
        {
            var message = new EmailMessage
            {
                Subject = subject,
                Body = body,
                To = recipients,
                //Cc = ccRecipients,
                //Bcc = bccRecipients
            };
            await Email.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException fbsEx)
        {
            // Email is not supported on this device
        }
        catch (Exception ex)
        {
            // Some other exception occurred
        }
    }
}
```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No platform differences.

## API

- [Email source code](#)
- [Email API documentation](#)

# Xamarin.Essentials: File System Helpers

11/13/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **FileSystem** class contains a series of helpers to find the application's cache and data directories and open files inside of the app package.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using File System Helpers

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To get the application's directory to store **cache data**. Cache data can be used for any data that needs to persist longer than temporary data, but should not be data that is required to properly operate.

```
var cacheDir = FileSystem.CacheDirectory;
```

To get the application's top-level directory for any files that are not user data files. These files are backed up with the operating system syncing framework. See Platform Implementation Specifics below.

```
var mainDir = FileSystem.AppDataDirectory;
```

To open a file that is bundled into the application package:

```
using (var stream = await FileSystem.OpenAppPackageFileAsync(templateFileName))
{
    using (var reader = new StreamReader(stream))
    {
        var fileContents = await reader.ReadToEndAsync();
    }
}
```

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)
- **CacheDirectory** – Returns the [CacheDir](#) of the current context.

- **AppDataDirectory** – Returns the [FilesDir](#) of the current context and are backed up using [Auto Backup](#) starting on API 23 and above.

Add any file into the **Assets** folder in the Android project and mark the Build Action as **AndroidAsset** to use it with `OpenAppPackageFileAsync`.

## API

- [File System Helpers source code](#)
- [File System API documentation](#)

# Xamarin.Essentials: Flashlight

11/13/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Flashlight** class has the ability to turn on or off the device's camera flash to turn it into a flashlight.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Flashlight** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The Flashlight and Camera permissions are required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Flashlight)]
[assembly: UsesPermission(Android.Manifest.Permission.Camera)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-permission android:name="android.permission.CAMERA" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **FLASHLIGHT** and **CAMERA** permissions. This will automatically update the **AndroidManifest.xml** file.

By adding these permissions [Google Play will automatically filter out devices](#) without specific hardware. You can get around this by adding the following to your AssemblyInfo.cs file in your Android project:

```
[assembly: UsesFeature("android.hardware.camera", Required = false)]
[assembly: UsesFeature("android.hardware.camera.autofocus", Required = false)]
```

## Using Flashlight

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The flashlight can be turned on and off through the `TurnOnAsync` and `TurnOffAsync` methods:

```
try
{
    // Turn On
    await Flashlight.TurnOnAsync();

    // Turn Off
    await Flashlight.TurnOffAsync();
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to turn on/off flashlight
}
```

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

The Flashlight class has been optimized based on the device's operating system.

### **API Level 23 and Higher**

On newer API levels, [Torch Mode](#) will be used to turn on or off the flash unit of the device.

### **API Level 22 and Lower**

A camera surface texture is created to turn on or off the `FlashMode` of the camera unit.

## API

- [Flashlight source code](#)
- [Flashlight API documentation](#)

# Xamarin.Essentials: Geocoding

11/11/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Geocoding** class provides APIs to geocode a placemark to a positional coordinates and reverse geocode coordinates to a placemark.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Geocoding** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

No additional setup required.

## Using Geocoding

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Getting [location](#) coordinates for an address:

```
try
{
    var address = "Microsoft Building 25 Redmond WA USA";
    var locations = await Geocoding.GetLocationsAsync(address);

    var location = locations?.FirstOrDefault();
    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

The altitude isn't always available. If it is not available, the `Altitude` property might be `null` or the value might be

zero. If the altitude is available, the value is in meters above sea level.

Getting [placemarks](#) for an existing set of coordinates:

```
try
{
    var lat = 47.673988;
    var lon = -122.121513;

    var placemarks = await Geocoding.GetPlacemarksAsync(lat, lon);

    var placemark = placemarks?.FirstOrDefault();
    if (placemark != null)
    {
        var geocodeAddress =
            $"AdminArea: {placemark.AdminArea}\n" +
            $"CountryCode: {placemark.CountryCode}\n" +
            $"CountryName: {placemark.CountryName}\n" +
            $"FeatureName: {placemark.FeatureName}\n" +
            $"Locality: {placemark.Locality}\n" +
            $"PostalCode: {placemark.PostalCode}\n" +
            $"SubAdminArea: {placemark.SubAdminArea}\n" +
            $"SubLocality: {placemark.SubLocality}\n" +
            $"SubThoroughfare: {placemark.SubThoroughfare}\n" +
            $"Thoroughfare: {placemark.Thoroughfare}\n";

        Console.WriteLine(geocodeAddress);
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

## Distance between Two Locations

The [Location](#) and [LocationExtensions](#) classes define methods to calculate the distance between two locations. See the article [Xamarin.Essentials: Geolocation](#) for an example.

## API

- [Geocoding source code](#)
- [Geocoding API documentation](#)

# Xamarin.Essentials: Geolocation

11/11/2018 • 3 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Geolocation** class provides APIs to retrieve the device's current geolocation coordinates.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Geolocation** functionality, the following platform-specific setup is required:

- [Android](#)
- [iOS](#)
- [UWP](#)

Coarse and Fine Location permissions are required and must be configured in the Android project. Additionally, if your app targets Android 5.0 (API level 21) or higher, you must declare that your app uses the hardware features in the manifest file. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessCoarseLocation)]
[assembly: UsesPermission(Android.Manifest.Permission.AccessFineLocation)]
[assembly: UsesFeature("android.hardware.location", Required = false)]
[assembly: UsesFeature("android.hardware.location.gps", Required = false)]
[assembly: UsesFeature("android.hardware.location.network", Required = false)]
```

Or update the Android manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name="android.hardware.location" android:required="false" />
<uses-feature android:name="android.hardware.location.gps" android:required="false" />
<uses-feature android:name="android.hardware.location.network" android:required="false" />
```

Or right-click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **ACCESS\_COARSE\_LOCATION** and **ACCESS\_FINE\_LOCATION** permissions. This will automatically update the **AndroidManifest.xml** file.

## Using Geolocation

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Geolocation API will also prompt the user for permissions when necessary.

You can get the last known [location](#) of the device by calling the `GetLastKnownLocationAsync` method. This is often faster then doing a full query, but can be less accurate.

```
try
{
    var location = await Geolocation.GetLastKnownLocationAsync();

    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

The altitude isn't always available. If it is not available, the `Altitude` property might be `null` or the value might be zero. If the altitude is available, the value is in meters above sea level.

To query the current device's [location](#) coordinates, the `GetLocationAsync` can be used. It is best to pass in a full `GeolocationRequest` and `CancellationToken` since it may take some time to get the device's location.

```
try
{
    var request = new GeolocationRequest(GeolocationAccuracy.Medium);
    var location = await Geolocation.GetLocationAsync(request);

    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

# Geolocation Accuracy

The following table outlines accuracy per platform:

## Lowest

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	3000
UWP	1000 - 5000

## Low

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	1000
UWP	300 - 3000

## Medium (default)

PLATFORM	DISTANCE (IN METERS)
Android	100 - 500
iOS	100
UWP	30-500

## High

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	10
UWP	<= 10

## Best

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	~0
UWP	<= 10

## Distance between Two Locations

The `Location` and `LocationExtensions` classes define `CalculateDistance` methods that allow you to calculate the distance between two geographic locations. This calculated distance does not take roads or other pathways into account, and is merely the shortest distance between the two points along the surface of the Earth, also known as the *great-circle distance* or colloquially, the distance "as the crow flies."

Here's an example:

```
Location boston = new Location(42.358056, -71.063611);
Location sanFrancisco = new Location(37.783333, -122.416667);
double miles = Location.CalculateDistance(boston, sanFrancisco, DistanceUnits.Miles);
```

The `Location` constructor has latitude and longitude arguments in that order. Positive latitude values are north of the equator, and positive longitude values are east of the Prime Meridian. Use the final argument to `CalculateDistance` to specify miles or kilometers. The `Location` class also defines `KilometersToMiles` and `MilesToKilometers` methods for converting between the two units.

## API

- [Geolocation source code](#)
- [Geolocation API documentation](#)

# Xamarin.Essentials: Gyroscope

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Gyroscope** class lets you monitor the device's gyroscope sensor which is the rotation around the device's three primary axes.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Gyroscope

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Gyroscope functionality works by calling the `Start` and `Stop` methods to listen for changes to the gyroscope. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```

public class GyroscopeTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public GyroscopeTest()
    {
        // Register for reading changes.
        Gyroscope.ReadingChanged += Gyroscope_ReadingChanged;
    }

    void Gyroscope_ReadingChanged(object sender, GyroscopeChangedEventArgs e)
    {
        var data = e.Reading;
        // Process Angular Velocity X, Y, and Z
        Console.WriteLine($"Reading: X: {data.AngularVelocity.X}, Y: {data.AngularVelocity.Y}, Z: {data.AngularVelocity.Z}");
    }

    public void ToggleGyroscope()
    {
        try
        {
            if (Gyroscope.IsMonitoring)
                Gyroscope.Stop();
            else
                Gyroscope.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}

```

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the [MainThread.BeginInvokeOnMainThread](#) method to run that code on the UI thread.

## API

- [Gyroscope source code](#)
- [Gyroscope API documentation](#)

# Xamarin.Essentials: Launcher

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Launcher** class enables an application to open a URI by the system. This is often used when deep linking into another application's custom URI schemes. If you are looking to open the browser to a website then you should refer to the [Browser API](#).

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Launcher

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To use the Launcher functionality call the `OpenAsync` method and pass in a `string` or `Uri` to open. Optionally, the `CanOpenAsync` method can be used to check if the URI schema can be handled by an application on the device.

```
public class LauncherTest
{
    public async Task OpenRideShareAsync()
    {
        var supportsUri = await Launcher.CanOpenAsync("lyft://");
        if (supportsUri)
            await Launcher.OpenAsync("lyft://ridetype?id=lyft_line");
    }
}
```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

The Task returned from `CanOpenAsync` completes immediately.

## API

- [Launcher source code](#)
- [Launcher API documentation](#)

# Xamarin.Essentials: Magnetometer

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Magnetometer** class lets you monitor the device's magnetometer sensor which indicates the device's orientation relative to Earth's magnetic field.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Magnetometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Magnetometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the magnetometer. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```

public class MagnetometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public MagnetometerTest()
    {
        // Register for reading changes.
        Magnetometer.RadingChanged += Magnetometer_ReadingChanged;
    }

    void Magnetometer_ReadingChanged(object sender, MagnetometerChangedEventArgs e)
    {
        var data = e.Rading;
        // Process MagneticField X, Y, and Z
        Console.WriteLine($"Reading: X: {data.MagneticField.X}, Y: {data.MagneticField.Y}, Z: {data.MagneticField.Z}");
    }

    public void ToggleMagnetometer()
    {
        try
        {
            if (Magnetometer.IsMonitoring)
                Magnetometer.Stop();
            else
                Magnetometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}

```

All data is returned in microteslas.

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the [MainThread.BeginInvokeOnMainThread](#) method to run that code on the UI thread.

## API

- [Magnetometer source code](#)
- [Magnetometer API documentation](#)

# Xamarin.Essentials: MainThread

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **MainThread** class allows applications to run code on the main thread of execution, and to determine if a particular block of code is currently running on the main thread.

## Background

Most operating systems — including iOS, Android, and the Universal Windows Platform — use a single-threading model for code involving the user interface. This model is necessary to properly serialize user-interface events, including keystrokes and touch input. This thread is often called the *main thread* or the *user-interface thread* or the *UI thread*. The disadvantage of this model is that all code that accesses user interface elements must run on the application's main thread.

Applications sometimes need to use events that call the event handler on a secondary thread of execution. (The Xamarin.Essentials classes `Accelerometer`, `Compass`, `Gyroscope`, `Magnetometer`, and `OrientationSensor` all might return information on a secondary thread when used with faster speeds.) If the event handler needs to access user-interface elements, it must run that code on the main thread. The **MainThread** class allows the application to run this code on the main thread.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Running Code on the Main Thread

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To run code on the main thread, call the static `MainThread.BeginInvokeOnMainThread` method. The argument is an `Action` object, which is simply a method with no arguments and no return value:

```
MainThread.BeginInvokeOnMainThread(() =>
{
    // Code to run on the main thread
});
```

It is also possible to define a separate method for the code that must run on the main thread:

```
void MyMainThreadCode()
{
    // Code to run on the main thread
}
```

You can then run this method on the main thread by referencing it in the `BeginInvokeOnMainThread` method:

```
MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
```

#### NOTE

Xamarin.Forms has a method called `Device.BeginInvokeOnMainThread(Action)` that does the same thing as `MainThread.BeginInvokeOnMainThread(Action)`. While you can use either method in a Xamarin.Forms app, consider whether or not the calling code has any other need for a dependency on Xamarin.Forms. If not, `MainThread.BeginInvokeOnMainThread(Action)` is likely a better option.

## Determining if Code is Running on the Main Thread

The `MainThread` class also allows an application to determine if a particular block of code is running on the main thread. The `IsMainThread` property returns `true` if the code calling the property is running on the main thread. A program can use this property to run different code for the main thread or a secondary thread:

```
if (MainThread.IsMainThread)
{
    // Code to run if this is the main thread
}
else
{
    // Code to run if this is a secondary thread
}
```

You might wonder if you should check if code is running on a secondary thread before calling `BeginInvokeOnMainThread`, for example, like this:

```
if (MainThread.IsMainThread)
{
    MyMainThreadCode();
}
else
{
    MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
}
```

You might suspect that this check might improve performance if the block of code is already running on the main thread.

*However, this check is not necessary.* The platform implementations of `BeginInvokeOnMainThread` themselves check if the call is made on the main thread. There is very little performance penalty if you call `BeginInvokeOnMainThread` when it's not really necessary.

## API

- [MainThread source code](#)
- [MainThread API documentation](#)



# Xamarin.Essentials: Maps

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Maps** class enables an application to open the installed maps application to a specific location or placemark.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Maps

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Maps functionality works by calling the `OpenAsync` method with the `Location` or `Placemark` to open with optional `MapsLaunchOptions`.

```
public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options = new MapsLaunchOptions { Name = "Microsoft Building 25" };

        await Maps.OpenAsync(location, options);
    }
}
```

When opening with a `Placemark`, the following information is required:

- `CountryName`
- `AdminArea`
- `Thoroughfare`
- `Locality`

```

public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var placemark = new Placemark
        {
            CountryName = "United States",
            AdminArea = "WA",
            Thoroughfare = "Microsoft Building 25",
            Locality = "Redmond"
        };
        var options = new MapsLaunchOptions { Name = "Microsoft Building 25" };

        await Maps.OpenAsync(placemark, options);
    }
}

```

## Extension Methods

If you already have a reference to a `Location` or `Placemark`, you can use the built-in extension method `OpenMapsAsync` with optional `MapsLaunchOptions`:

```

public class MapsTest
{
    public async Task OpenPlacemarkOnMaps(Placemark placemark)
    {
        await placemark.OpenMapsAsync();
    }
}

```

## Directions Mode

If you call `OpenMapsAsync` without any `MapsLaunchOptions`, the map will launch to the location specified. Optionally, you can have a navigation route calculated from the device's current position. This is accomplished by setting the `MapDirectionsMode` on the `MapsLaunchOptions`:

```

public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options = new MapsLaunchOptions { MapDirectionsMode = MapDirectionsMode.Driving };

        await Maps.OpenAsync(location, options);
    }
}

```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- `MapDirectionsMode` supports Bicycling, Driving, and Walking.

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

Android uses the `geo:` Uri scheme to launch the maps application on the device. This may prompt the user to select from an existing app that supports this Uri scheme. Xamarin.Essentials is tested with Google Maps, which supports this scheme.

## API

- [Maps source code](#)
- [Maps API documentation](#)

# Xamarin.Essentials: Browser

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Browser** class enables an application to open a web link in the optimized system preferred browser or the external browser.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Browser

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Browser functionality works by calling the `OpenAsync` method with the `Uri` and `BrowserLaunchMode`.

```
public class BrowserTest
{
    public async Task OpenBrowser(Uri uri)
    {
        await Browser.OpenAsync(uri, BrowserLaunchMode.SystemPreferred);
    }
}
```

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

The Launch Mode determines how the browser is launched:

## System Preferred

[Chrome Custom Tabs](#) will attempt to be used load the Uri and keep navigation awareness.

## External

An `Intent` will be used to request the Uri be opened through the systems normal browser.

## API

- Browser source code
- Browser API documentation

# Xamarin.Essentials: OrientationSensor

10/31/2018 • 3 minutes to read • [Edit Online](#)



Pre-release NuGet

The **OrientationSensor** class lets you monitor the orientation of a device in three dimensional space.

## NOTE

This class is for determining the orientation of a device in 3D space. If you need to determine if the device's video display is in portrait or landscape mode, use the `Orientation` property of the `ScreenMetrics` object available from the `DeviceDisplay` class.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using OrientationSensor

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The `OrientationSensor` is enabled by calling the `Start` method to monitor changes to the device's orientation, and disabled by calling the `Stop` method. Any changes are sent back through the `ReadingChanged` event. Here is a sample usage:

```

public class OrientationSensorTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public OrientationSensorTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        OrientationSensor.RadingChanged += OrientationSensor_RadingChanged;
    }

    void OrientationSensor_RadingChanged(object sender, OrientationSensorChangedEventArgs e)
    {
        var data = e.Rading;
        Console.WriteLine($"Rading: X: {data.Orientation.X}, Y: {data.Orientation.Y}, Z: {data.Orientation.Z}, W: {data.Orientation.W}");
        // Process Orientation quaternion (X, Y, Z, and W)
    }

    public void ToggleOrientationSensor()
    {
        try
        {
            if (OrientationSensor.IsMonitoring)
                OrientationSensor.Stop();
            else
                OrientationSensor.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}

```

`OrientationSensor` readings are reported back in the form of a `Quaternion` that describes the orientation of the device based on two 3D coordinate systems:

The device (generally a phone or tablet) has a 3D coordinate system with the following axes:

- The positive X axis points to the right of the display in portrait mode.
- The positive Y axis points to the top of the device in portrait mode.
- The positive Z axis points out of the screen.

The 3D coordinate system of the Earth has the following axes:

- The positive X axis is tangent to the surface of the Earth and points east.
- The positive Y axis is also tangent to the surface of the Earth and points north.
- The positive Z axis is perpendicular to the surface of the Earth and points up.

The `Quaternion` describes the rotation of the device's coordinate system relative to the Earth's coordinate system.

A `Quaternion` value is very closely related to rotation around an axis. If an axis of rotation is the normalized vector  $(a_x, a_y, a_z)$ , and the rotation angle is  $\Theta$ , then the  $(X, Y, Z, W)$  components of the quaternion are:

$$(a_x \cdot \sin(\Theta/2), a_y \cdot \sin(\Theta/2), a_z \cdot \sin(\Theta/2), \cos(\Theta/2))$$

These are right-hand coordinate systems, so with the thumb of the right hand pointed in the positive direction of

the rotation axis, the curve of the fingers indicate the direction of rotation for positive angles.

Examples:

- When the device lies flat on a table with its screen facing up, with the top of the device (in portrait mode) pointing north, the two coordinate systems are aligned. The `Quaternion` value represents the identity quaternion (0, 0, 0, 1). All rotations can be analyzed relative to this position.
- When the device lies flat on a table with its screen facing up, and the top of the device (in portrait mode) pointing west, the `Quaternion` value is (0, 0, 0.707, 0.707). The device has been rotated 90 degrees around the Z axis of the Earth.
- When the device is held upright so that the top (in portrait mode) points towards the sky, and the back of the device faces north, the device has been rotated 90 degrees around the X axis. The `Quaternion` value is (0.707, 0, 0, 0.707).
- If the device is positioned so its left edge is on a table, and the top points north, the device has been rotated -90 degrees around the Y axis (or 90 degrees around the negative Y axis). The `Quaternion` value is (0, -0.707, 0, 0.707).

## Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Normal** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

## API

- [OrientationSensor source code](#)
- [OrientationSensor API documentation](#)

# Xamarin.Essentials: Phone Dialer

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **PhoneDialer** class enables an application to open a phone number in the dialer.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Phone Dialer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Phone Dialer functionality works by calling the `open` method with a phone number to open the dialer with.

When `open` is requested the API will automatically attempt to format the number based on the country code if specified.

```
public class PhoneDialerTest
{
    public async Task PlacePhoneCall(string number)
    {
        try
        {
            PhoneDialer.Open(number);
        }
        catch (ArgumentNullException anEx)
        {
            // Number was null or white space
        }
        catch (FeatureNotSupportedException ex)
        {
            // Phone Dialer is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

## API

- [Phone Dialer source code](#)
- [Phone Dialer API documentation](#)

# Xamarin.Essentials: Power Energy Saver Status

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Power** class provides information about the device's energy-saver status, which indicates if the device is running in a low-power mode. Applications should avoid background processing if the device's energy-saver status is on.

## Background

Devices that run on batteries can be put into a low-power energy-saver mode. Sometimes devices are switched into this mode automatically, for example, when the battery drops below 20% capacity. The operating system responds to energy-saver mode by reducing activities that tend to deplete the battery. Applications can help by avoiding background processing or other high-power activities when energy-saver mode is on.

For Android devices, the **Power** class returns meaningful information only for Android version 5.0 (Lollipop) and above.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using the Power class

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Obtain the current energy-saver status of the device using the static `Power.EnergySaverStatus` property:

```
// Get energy saver status
var status = Power.EnergySaverStatus;
```

This property returns a member of the `EnergySaverStatus` enumeration, which is either `On`, `Off`, or `Unknown`. If the property returns `On`, the application should avoid background processing or other activities that might consume a lot of power.

The application should also install an event handler. The **Power** class exposes an event that is triggered when the energy-saver status changes:

```
public class EnergySaverTest
{
    public EnergySaverTest()
    {
        // Subscribe to changes of energy-saver status
        Power.EnergySaverStatusChanged += OnEnergySaverStatusChanged;
    }

    private void OnEnergySaverStatusChanged(EnergySaverStatusChangedEventArgs e)
    {
        // Process change
        var status = e.EnergySaverStatus;
    }
}
```

If the energy-saver status changes to `On`, the application should stop performing background processing. If the status changes to `Unknown` or `Off`, the application can resume background processing.

## API

- [Power source code](#)
- [Power API documentation](#)

# Xamarin.Essentials: Preferences

11/13/2018 • 2 minutes to read • [Edit Online](#)



The **Preferences** class helps to store application preferences in a key/value store.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Preferences

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To save a value for a given *key* in preferences:

```
Preferences.Set("my_key", "my_value");
```

To retrieve a value from preferences or a default if not set:

```
var myValue = Preferences.Get("my_key", "default_value");
```

To remove the *key* from preferences:

```
Preferences.Remove("my_key");
```

To remove all preferences:

```
Preferences.Clear();
```

In addition to these methods each take in an optional `sharedName` that can be used to create additional containers for preference. Read the platform implementation specifics below.

## Supported Data Types

The following data types are supported in **Preferences**:

- **bool**
- **double**
- **int**

- [float](#)
- [long](#)
- [string](#)
- [DateTime](#)

## Implementation Details

Values of `DateTime` are stored in a 64-bit binary (long integer) format using two methods defined by the `DateTime` class: The `ToBinary` method is used to encode the `DateTime` value, and the `FromBinary` method decodes the value. See the documentation of these methods for adjustments that might be made to decoded values when a `DateTime` is stored that is not a Coordinated Universal Time (UTC) value.

## Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

All data is stored into [Shared Preferences](#). If no `sharedName` is specified the default shared preferences are used, else the name is used to get a **private** shared preferences with the specified name.

## Persistence

Uninstalling the application will cause all *Preferences* to be removed. There is one exception to this, which for apps that target and run on Android 6.0 (API level 23) or later that are using [Auto Backup](#). This feature is on by default and preserves app data including **Shared Preferences**, which is what the **Preferences** API utilizes. You can disable this by following Google's [documentation](#).

## Limitations

When storing a string, this API is intended to store small amounts of text. Performance may be subpar if you try to use it to store large amounts of text.

## API

- [Preferences source code](#)
- [Preferences API documentation](#)

# Xamarin.Essentials: Screen Lock

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **ScreenLock** class can request to keep the screen from falling asleep when the application is running.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using ScreenLock

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The screen lock functionality works by calling the `RequestActive` and `RequestRelease` methods to request the screen from turning off.

```
public class ScreenLockTest
{
    public void ToggleScreenLock()
    {
        if (!ScreenLock.IsActive)
            ScreenLock.RequestActive();
        else
            ScreenLock.RequestRelease();
    }
}
```

## API

- [Screen Lock source code](#)
- [Screen Lock API documentation](#)

# Xamarin.Essentials: Secure Storage

10/31/2018 • 3 minutes to read • [Edit Online](#)



Pre-release NuGet

The **SecureStorage** class helps securely store simple key/value pairs.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **SecureStorage** functionality, the following platform-specific setup is required:

- [Android](#)
- [iOS](#)
- [UWP](#)

### TIP

[Auto Backup for Apps](#) is a feature of Android 6.0 (API level 23) and later that backs up user's app data (shared preferences, files in the app's internal storage, and other specific files). Data is restored when an app is re-installed or installed on a new device. This can impact `SecureStorage` which utilizes share preferences that are backed up and can not be decrypted when the restore occurs. Xamarin.Essentials automatically handles this case by removing the key so it can be reset, but you can take an additional step by disabling Auto Backup.

## Enable or disable backup

You can choose to disable Auto Backup for your entire application by setting the `android:allowBackup` setting to false in the `AndroidManifest.xml` file. This approach is only recommended if you plan on restoring data in another way.

```
<manifest ... >
  ...
  <application android:allowBackup="false" ... >
    ...
  </application>
</manifest>
```

## Selective Backup

Auto Backup can be configured to disable specific content from backing up. You can create a custom rule set to exclude `SecureStore` items from being backed up.

1. Set the `android:fullBackupContent` attribute in your **AndroidManifest.xml**:

```
<application ...
  android:fullBackupContent="@xml/auto_backup_rules">
</application>
```

2. Create a new XML file named **auto\_backup\_rules.xml** in the **Resources/xml** directory. Then set the following content that includes all shared preferences except for `SecureStorage`:

```
<?xml version="1.0" encoding="utf-8"?>
<full-backup-content>
    <include domain="sharedpref" path="."/> 
    <exclude domain="sharedpref" path="${applicationId}.xamarinessentials.xml"/>
</full-backup-content>
```

## Using Secure Storage

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

To save a value for a given *key* in secure storage:

```
try
{
    await SecureStorage.SetAsync("oauth_token", "secret-oauth-token-value");
}
catch (Exception ex)
{
    // Possible that device doesn't support secure storage on device.
}
```

To retrieve a value from secure storage:

```
try
{
    var oauthToken = await SecureStorage.GetAsync("oauth_token");
}
catch (Exception ex)
{
    // Possible that device doesn't support secure storage on device.
}
```

### NOTE

If there is no value associated with the requested key, `GetAsync` will return `null`.

To remove a specific key, call:

```
SecureStorage.Remove("oauth_token");
```

To remove all keys, call:

```
SecureStorage.RemoveAll();
```

## Platform Implementation Specifics

- [Android](#)

- [iOS](#)
- [UWP](#)

The [Android KeyStore](#) is used to store the cipher key used to encrypt the value before it is saved into a [Shared Preferences](#) with a filename of **[YOUR-APP-PACKAGE-ID].xamarinessentials**. The key used in the shared preferences file is a *MD5 Hash* of the key passed into the `SecureStorage` APIs.

## API Level 23 and Higher

On newer API levels, an **AES** key is obtained from the Android KeyStore and used with an **AES/GCM/NoPadding** cipher to encrypt the value before it is stored in the shared preferences file.

## API Level 22 and Lower

On older API levels, the Android KeyStore only supports storing **RSA** keys, which is used with an **RSA/ECB/PKCS1Padding** cipher to encrypt an **AES** key (randomly generated at runtime) and stored in the shared preferences file under the key *SecureStorageKey*, if one has not already been generated.

**SecureStorage** uses the [Preferences API](#) and follows the same data persistence outlined in the [Preferences documentation](#). If a device upgrades from API level 22 or lower to API level 23 and higher, this type of encryption will continue to be used unless the app is uninstalled or **RemoveAll** is called.

## Limitations

This API is intended to store small amounts of text. Performance may be slow if you try to use it to store large amounts of text.

## API

- [SecureStorage source code](#)
- [SecureStorage API documentation](#)

# Xamarin.Essentials: SMS

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Sms** class enables an application to open the default SMS application with a specified message to send to a recipient.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Sms

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The SMS functionality works by calling the `ComposeAsync` method an `SmsMessage` that contains the message's recipient and the body of the message, both of which are optional.

```
public class SmsTest
{
    public async Task SendSms(string messageText, string recipient)
    {
        try
        {
            var message = new SmsMessage(messageText, recipient);
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Additionally, you can pass in multiple recipients to a `SmsMessage`:

```
public class SmsTest
{
    public async Task SendSms(string messageText, string[] recipients)
    {
        try
        {
            var message = new SmsMessage(messageText, recipients);
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

## API

- [Sms source code](#)
- [Sms API documentation](#)

# Xamarin.Essentials: Text-to-Speech

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **TextToSpeech** class enables an application to utilize the built-in text-to-speech engines to speak back text from the device and also to query available languages that the engine can support.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Text-to-Speech

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Text-to-Speech works by calling the `speakAsync` method with text and optional parameters, and returns after the utterance has finished.

```
public async Task SpeakNowDefaultSettings()
{
    await TextToSpeech.SpeakAsync("Hello World");

    // This method will block until utterance finishes.
}

public void SpeakNowDefaultSettings2()
{
    TextToSpeech.SpeakAsync("Hello World").ContinueWith((t) =>
    {
        // Logic that will run after utterance finishes.

    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

This method takes in an optional `CancellationToken` to stop the utterance once it starts.

```

CancellationTokenSource cts;
public async Task SpeakNowDefaultSettings()
{
    cts = new CancellationTokenSource();
    await TextToSpeech.SpeakAsync("Hello World", cancelToken: cts.Token);

    // This method will block until utterance finishes.
}

public void CancelSpeech()
{
    if (cts?.IsCancellationRequested ?? false)
        return;

    cts.Cancel();
}

```

Text-to-Speech will automatically queue speech requests from the same thread.

```

bool isBusy = false;
public void SpeakMultiple()
{
    isBusy = true;
    Task.Run(async () =>
    {
        await TextToSpeech.SpeakAsync("Hello World 1");
        await TextToSpeech.SpeakAsync("Hello World 2");
        await TextToSpeech.SpeakAsync("Hello World 3");
        isBusy = false;
    });

    // or you can query multiple without a Task:
    Task.WhenAll(
        TextToSpeech.SpeakAsync("Hello World 1"),
        TextToSpeech.SpeakAsync("Hello World 2"),
        TextToSpeech.SpeakAsync("Hello World 3"))
        .ContinueWith((t) => { isBusy = false; }, TaskScheduler.FromCurrentSynchronizationContext());
}

```

## Speech Settings

For more control over how the audio is spoken back with `SpeakSettings` that allows setting the volume, pitch, and locale.

```

public async Task SpeakNow()
{
    var settings = new SpeakSettings()
    {
        Volume = .75,
        Pitch = 1.0
    };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}

```

The following are supported values for these parameters:

PARAMETER	MINIMUM	MAXIMUM
Pitch	0	2.0

PARAMETER	MINIMUM	MAXIMUM
Volume	0	1.0

## Speech Locales

Each platform supports different locales, to speak back text in different languages and accents. Platforms have different codes and ways of specifying the locale, which is why `Xamarin.Essentials` provides a cross-platform `Locale` class and a way to query them with `GetLocalesAsync`.

```
public async Task SpeakNow()
{
    var locales = await TextToSpeech.GetLocalesAsync();

    // Grab the first locale
    var locale = locales.FirstOrDefault();

    var settings = new SpeakSettings()
    {
        Volume = .75,
        Pitch = 1.0,
        Locale = locale
    };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}
```

## Limitations

- Utterance queue is not guaranteed if called across multiple threads.
- Background audio playback is not officially supported.

## API

- [TextToSpeech source code](#)
- [TextToSpeech API documentation](#)

# Xamarin.Essentials: Version Tracking

10/31/2018 • 2 minutes to read • [Edit Online](#)



The **VersionTracking** class lets you check the applications version and build numbers along with seeing additional information such as if it is the first time the application launched ever or for the current version, get the previous build information, and more.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

## Using Version Tracking

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The first time you use the **VersionTracking** class it will start tracking the current version. You must call `Track` early only in your application each time it is loaded to ensure the current version information is tracked:

```
VersionTracking.Track();
```

After the initial `Track` is called version information can be read:

```
// First time ever launched application
var firstLaunch = VersionTracking.IsFirstLaunchEver;

// First time launching current version
var firstLaunchCurrent = VersionTracking.IsFirstLaunchForCurrentVersion;

// First time launching current build
var firstLaunchBuild = VersionTracking.IsFirstLaunchForCurrentBuild;

// Current app version (2.0.0)
var currentVersion = VersionTracking.CurrentVersion;

// Current build (2)
var currentBuild = VersionTracking.CurrentBuild;

// Previous app version (1.0.0)
var previousVersion = VersionTracking.PreviousVersion;

// Previous app build (1)
var previousBuild = VersionTracking.PreviousBuild;

// First version of app installed (1.0.0)
var firstVersion = VersionTracking.FirstInstalledVersion;

// First build of app installed (1)
var firstBuild = VersionTracking.FirstInstalledBuild;

// List of versions installed (1.0.0, 2.0.0)
var versionHistory = VersionTracking.VersionHistory;

// List of builds installed (1, 2)
var buildHistory = VersionTracking.BuildHistory;
```

## Platform Implementation Specifics

All version information is stored using the [Preferences](#) API in Xamarin.Essentials and is stored with a filename of **[YOUR-APP-PACKAGE-ID].xamarinessentials.versiontracking** and follows the same data persistence outlined in the [Preferences](#) documentation.

## API

- [Version Tracking source code](#)
- [Version Tracking API documentation](#)

# Xamarin.Essentials: Vibration

10/31/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

The **Vibration** class lets you start and stop the vibrate functionality for a desired amount of time.

## Get started

To start using this API, read the [getting started](#) guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Vibration** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The Vibrate permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.VIBRATE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **VIBRATE** permission. This will automatically update the **AndroidManifest.xml** file.

## Using Vibration

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Vibration functionality can be requested for a set amount of time or the default of 500 milliseconds.

```
try
{
    // Use default vibration length
    Vibration.Vibrate();

    // Or use specified time
    var duration = TimeSpan.FromSeconds(1);
    Vibration.Vibrate(duration);
}

catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

Cancellation of device vibration can be requested with the `Cancel` method:

```
try
{
    Vibration.Cancel();
}

catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

## Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No platform differences.

## API

- [Vibration source code](#)
- [Vibration API documentation](#)

# Xamarin.Essentials: Troubleshooting

7/10/2018 • 2 minutes to read • [Edit Online](#)



Pre-release NuGet

## Error: Version conflict detected for Xamarin.Android.Support.Compat

The following error may occur when updating NuGet packages (or adding a new package) with a Xamarin.Forms project that uses Xamarin.Essentials:

```
NU1107: Version conflict detected for Xamarin.Android.Support.Compat. Reference the package directly from the
project to resolve this issue.
MyApp -> Xamarin.Essentials 0.8.0-preview -> Xamarin.Android.Support.CustomTabs 27.0.2.1 ->
Xamarin.Android.Support.Compat (= 27.0.2.1)
MyApp -> Xamarin.Forms 3.1.0.583944 -> Xamarin.Android.Support.v4 25.4.0.2 -> Xamarin.Android.Support.Compat
(= 25.4.0.2).
```

The problem is mismatched dependencies for the two NuGets. This can be resolved by manually adding a specific version of the dependency (in this case **Xamarin.Android.Support.Compat**) that can support both.

To do this, add the NuGet that is the source of the conflict manually, and use the **Version** list to select a specific version. Currently version 27.0.2.1 of the Xamarin.Android.Support.Compat & Xamarin.Android.Support.Core.Util NuGet will resolve this error.

Refer to [this blog post](#) for more information and a video on how to resolve the issue.

If run into any issues or find a bug please report it on the [Xamarin.Essentials GitHub repository](#).

# Data & Cloud Services

4/12/2018 • 2 minutes to read • [Edit Online](#)

*Xamarin.Forms applications can consume web services implemented using a wide variety of technologies, and this guide will examine how to do this.*

For an introduction to cross-platform web service consumption on the Xamarin platform, see [Introduction to Web Services](#).

## Understanding the Sample

This article provides a walkthrough of the Xamarin.Forms sample application that demonstrates how to communicate with different web services. Topics covered include the anatomy of the application, the pages, data model, and invoking web service operations.

## Consuming Web Services

This guide demonstrates how to communicate with different web services to provide create, read, update, and delete (CRUD) functionality to a Xamarin.Forms application. Topics covered include communicating with [ASMX services](#), [WCF services](#), [REST services](#), and [Azure Mobile Apps](#).

## Authenticating Access to Web Services

This guide explains how to integrate authentication services into a Xamarin.Forms application to enable users to share a backend while only having access to their own data. Topics covered include [using basic authentication with a REST service](#), [using the Xamarin.Auth component to authenticate against OAuth identity providers](#), and using the inbuilt authentication mechanisms offered by [Azure Mobile Apps](#).

## Synchronizing Data with Web Services

This article explains how to add offline sync functionality to a Xamarin.Forms application. Offline sync allows users to interact with a mobile application, viewing, adding, or modifying data, even where there isn't a network connection. Changes are stored in a local database, and once the device is online, the changes can be synced with the web service.

## Sending Push Notifications

This article demonstrates how to add push notifications to a Xamarin.Forms application. Azure Notification Hubs provide a scalable push infrastructure for sending mobile push notifications from any backend to any mobile platform, while eliminating the complexity of a backend having to communicate with different platform notification systems.

## Storing Files in the Cloud

This article demonstrates how to use Xamarin.Forms to store text and binary data in Azure Storage, and how to access the data. Azure Storage is a scalable cloud storage solution that can be used for storing unstructured, and structured data.

## Searching Data in the Cloud

This article demonstrates how to use the Microsoft Azure Search Library to integrate Azure Search into a Xamarin.Forms application. Azure Search is a cloud service that provides indexing and querying capabilities for uploaded data. This removes the infrastructure requirements and search algorithm complexities traditionally associated with implementing search functionality in an application.

## Storing Data in a Document Database

This guide demonstrates how to use the Azure Cosmos DB .NET Standard client library to integrate an Azure Cosmos DB document database into a Xamarin.Forms application. An Azure Cosmos DB document database is a NoSQL database that provides low latency access to JSON documents, offering a fast, highly available, scalable database service for applications that require seamless scale and global replication.

## Adding Intelligence with Cognitive Services

This guide explains how to use some of the Microsoft Cognitive Services APIs in a Xamarin.Forms application. Cognitive Services are a set of APIs, SDKs, and services available to developers to make their applications more intelligent by adding features such as facial recognition, speech recognition, and language understanding.

# Understanding the Sample

6/7/2018 • 4 minutes to read • [Edit Online](#)

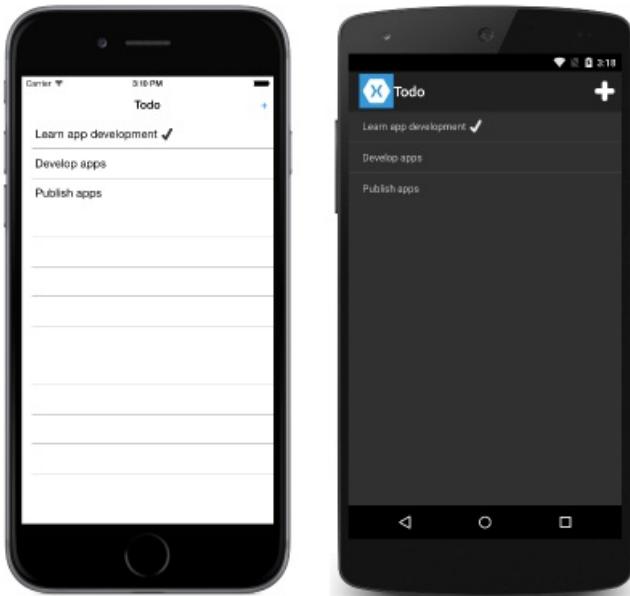
This topic provides a walkthrough of the `Xamarin.Forms` sample application that demonstrates how to communicate with different web services. While each web service uses a separate sample application, they are functionally similar and share common classes.

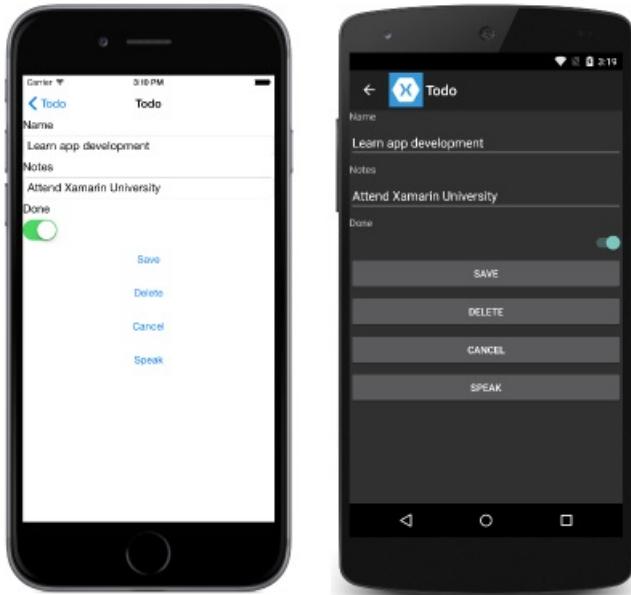
The sample to-do list application described below is used to demonstrate how to access different types of web service backends with `Xamarin.Forms`. It provides functionality to:

- View a list of tasks.
- Add, edit, and delete tasks.
- Set a task's status to 'done'.
- Speak the task's name and notes fields.

In all cases, the tasks are stored in a backend that's accessed through a web service.

When the application is launched, a page is displayed that lists any tasks retrieved from the web service, and allows the user to create a new task. Clicking on a task navigates the application to a second page where the task can be edited, saved, deleted, and spoken. The final application is shown below:





Each topic in this guide provides a download link to a *different* version of the application that demonstrates a specific type of web service backend. Download the relevant sample code on the page relating to each web-service style.

## Understanding the Application Anatomy

The PCL project for each sample application consists of three main folders:

FOLDER	PURPOSE
Data	Contains the classes and interfaces used to manage data items, and communicate with the web service. At a minimum, this includes the <code>TodoItemManager</code> class, which is exposed through a property in the <code>App</code> class to invoke web service operations.
Models	Contains the data model classes for the application. At a minimum, this includes the <code>TodoItem</code> class, which models a single item of data used by the application. The folder can also include any additional classes used to model user data.
Views	Contains the pages for the application. This usually consists of the <code>TodoListPage</code> and <code>TodoItemPage</code> classes, and any additional classes used for authentication purposes.

The PCL project for each application also consists of a number of important files:

FILE	PURPOSE
Constants.cs	The <code>Constants</code> class, which specifies any constants used by the application to communicate with the web service. These constants require updating to access your personal backend service created on a provider.
ITextToSpeech.cs	The <code>ITextToSpeech</code> interface, which specifies that the <code>Speak</code> method must be provided by any implementing classes.

FILE	PURPOSE
Todo.cs	The <code>App</code> class that is responsible for instantiating both the first page that will be displayed by the application on each platform, and the <code>TodoItemManager</code> class that is used to invoke web service operations.

## Viewing Pages

The majority of the sample applications contain at least two pages:

- **TodoListPage** – this page displays a list of `TodoItem` instances, and a tick icon if the `TodoItem.Done` property is `true`. Clicking on an item navigates to the `TodoItemPage`. In addition, new items can be created by clicking on the + symbol.
- **TodoItemPage** – this page displays the details for the selected `TodoItem`, and allows it to be edited, saved, deleted, and spoken.

In addition, some sample applications contain additional pages that are used to manage the user authentication process.

## Modeling the Data

Each sample application uses the `TodoItem` class to model the data that is displayed and sent to the web service for storage. The following code example shows the `TodoItem` class:

```
public class TodoItem
{
    public string ID { get; set; }
    public string Name { get; set; }
    public string Notes { get; set; }
    public bool Done { get; set; }
}
```

The `ID` property is used to uniquely identify each `TodoItem` instance, and is used by each web service to identify data to be updated or deleted.

## Invoking Web Service Operations

Web service operations are accessed through the `TodoItemManager` class, and an instance of the class can be accessed through the `App.TodoManager` property. The `TodoItemManager` class provides the following methods to invoke web service operations:

- **GetTasksAsync** – this method is used to populate the `ListView` control on the `TodoListPage` with the `TodoItem` instances retrieved from the web service.
- **SaveTaskAsync** – this method is used to create or update a `TodoItem` instance on the web service.
- **DeleteTaskAsync** – this method is used to delete a `TodoItem` instance on the web service.

In addition, some sample applications contain additional methods in the `TodoItemManager` class, which are used to manage the user authentication process.

Rather than invoke the web service operations directly, the `TodoItemManager` methods invoke methods on a dependent class that is injected into the `TodoItemManager` constructor. For example, one sample application injects the `RestService` class into the `TodoItemManager` constructor to provide the implementation that uses REST APIs to access data.

## Translating Text to Speech

The majority of the sample applications contain text-to-speech (TTS) functionality to speak the values of the `TodoItem.Name` and `TodoItem.Notes` properties. This is accomplished by the `OnSpeakActivated` event handler in the

`TodoItemPage` class, as shown in the following code example:

```
void OnSpeakActivated (object sender, EventArgs e)
{
    var todoItem = (TodoItem)BindingContext;
    App.Speech.Speak(todoItem.Name + " " + todoItem.Notes);
}
```

This method simply invokes the `Speak` method that is implemented by a platform-specific `Speech` class. Each `Speech` class implements the `ITextToSpeech` interface, and platform-specific startup code creates an instance of the `Speech` class that can be accessed through the `App.Speech` property.

## Summary

This topic provided a walkthrough of the Xamarin.Forms sample application that's used to demonstrate how to communicate with different web services. While each web service uses a separate sample application, they are all based on the same user-interface and business logic as described above - only the web service data storage mechanism is different.

## Related Links

- [ASMX version \(sample\)](#)
- [WCF version \(sample\)](#)
- [REST version \(sample\)](#)
- [Azure version \(sample\)](#)

# Consuming Web Services

4/12/2018 • 2 minutes to read • [Edit Online](#)

This guide demonstrates how to communicate with different web services to provide create, read, update, and delete (CRUD) functionality to a Xamarin.Forms application. Topics covered include communicating with ASMX services, WCF services, REST services, and Azure Mobile Apps.

## Consuming an ASP.NET Web Service (ASMX)

ASP.NET Web Services (ASMX) provide the ability to build web services that send messages over HTTP using Simple Object Access Protocol (SOAP). SOAP is a platform-independent and language-independent protocol for building and accessing web services. Consumers of an ASMX service do not need to know anything about the platform, object model, or programming language used to implement the service. They only need to understand how to send and receive SOAP messages. This article demonstrates how to consume an ASMX web service from a Xamarin.Forms application.

## Consuming a Windows Communication Foundation (WCF) Web Service

WCF is Microsoft's unified framework for building service-oriented applications. It enables developers to build secure, reliable, transacted, and interoperable distributed applications. There are differences between ASP.NET Web Services (ASMX) and WCF, but it is important to understand that WCF supports the same capabilities that ASMX provides — SOAP messages over HTTP. This article demonstrates how to consume an WCF SOAP service from a Xamarin.Forms application.

## Consuming a RESTful Web Service

Representational State Transfer (REST) is an architectural style for building web services. REST requests are made over HTTP using the same HTTP verbs that web browsers use to retrieve web pages and to send data to servers. This article demonstrates how to consume a RESTful web service from a Xamarin.Forms application.

## Consuming an Azure Mobile App

Azure Mobile Apps allow you to develop apps with scalable backends hosted in Azure App Service, with support for mobile authentication, offline sync, and push notifications. This article, which is only applicable to Azure Mobile Apps that use a Node.js backend, explains how to query, insert, update, and delete data stored in a table in an Azure Mobile Apps instance.

## Related Links

- [Introduction to Web Services](#)
- [Async Support Overview](#)

# Consuming an ASP.NET Web Service (ASMX)

6/8/2018 • 8 minutes to read • [Edit Online](#)

ASMX provides the ability to build web services that send messages using the Simple Object Access Protocol (SOAP). SOAP is a platform-independent and language-independent protocol for building and accessing web services. Consumers of an ASMX service do not need to know anything about the platform, object model, or programming language used to implement the service. They only need to understand how to send and receive SOAP messages. This article demonstrates how to consume an ASMX SOAP service from a Xamarin.Forms application.

A SOAP message is an XML document containing the following elements:

- A root element named *Envelope* that identifies the XML document as a SOAP message.
- An optional *Header* element that contains application-specific information such as authentication data. If the *Header* element is present it must be the first child element of the *Envelope* element.
- A required *Body* element that contains the SOAP message intended for the recipient.
- An optional *Fault* element that's used to indicate error messages. If the *Fault* element is present, it must be a child element of the *Body* element.

SOAP can operate over many transport protocols, including HTTP, SMTP, TCP, and UDP. However, an ASMX service can only operate over HTTP. The Xamarin platform supports standard SOAP 1.1 implementations over HTTP, and this includes support for many of the standard ASMX service configurations.

Instructions on setting up the ASMX service can be found in the readme file that accompanies the sample application. However, when the sample application is run, it will connect to a Xamarin-hosted ASMX service that provides read-only access to data, as shown in the following screenshot:

The screenshot shows a Xamarin Forms application interface. On the left, a code editor displays the `TodoService.asmx.cs` file, which contains C# code for an ASMX service. On the right, two smartphone emulators show the running application. Both screens display a "Todo" list with items like "Learn app development", "Develop apps", and "Publish apps". The code editor highlights a line of code related to the `TodoService` constructor.

```
TodoService.asmx.cs
TodoASMXService
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Web.Services;
using TodoASMXService.Models;
using TodoASMXService.Services;

namespace TodoASMXService
{
    public class TodoService : System.Web.Services.WebService
    {
        public TodoService()
        {
            s_["")];
            1]);
            s.TodoService(new TodoRepository());
        }

        [WebMethod]
        public string[] GetTodos()
        {
            return todos;
        }

        [WebMethod]
        public void AddTodo(string title)
        {
            todos.Add(title);
        }

        [WebMethod]
        public void RemoveTodo(string title)
        {
            todos.Remove(title);
        }
    }
}
```

#### NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

## Consuming the Web Service

The ASMX service provides the following operations:

OPERATION	DESCRIPTION	PARAMETERS
GetTodoItems	Get a list of to-do items	
CreateTodoItem	Create a new to-do item	An XML serialized TodoItem
EditTodoItem	Update a to-do item	An XML serialized TodoItem
DeleteTodoItem	Delete a to-do item	An XML serialized TodoItem

For more information about the data model used in the application, see [Modeling the data](#).

#### NOTE

The sample application consumes the Xamarin-hosted ASMX service that provides read-only access to the web service. Therefore, the operations that create, update, and delete data will not alter the data consumed in the application. However, a hostable version of the ASMX service is available in the **TodoASMXService** folder in the accompanying sample application. This hostable version of the ASMX service permits full create, update, read, and delete access to the data.

A proxy must be generated to consume the ASMX service, which allows the application to connect to the service. The proxy is constructed by consuming service metadata that defines the methods and associated service configuration. This metadata is exposed in the form of a Web Services Description Language (WSDL) document that is generated by the web service. The proxy is built by adding a web reference for the web service to the platform-specific projects.

The generated proxy classes provide methods for consuming the web service that use the Asynchronous Programming Model (APM) design pattern. In this pattern an asynchronous operation is implemented as two methods named *BeginOperationName* and *EndOperationName*, which begin and end the asynchronous operation.

The *BeginOperationName* method begins the asynchronous operation and returns an object that implements the `IAsyncResult` interface. After calling *BeginOperationName*, an application can continue executing instructions on the calling thread, while the asynchronous operation takes place on a thread pool thread.

For each call to *BeginOperationName*, the application should also call *EndOperationName* to get the results of the operation. The return value of *EndOperationName* is the same type returned by the synchronous web service method. For example, the `EndGetTodoItems` method returns a collection of `TodoItem` instances. The *EndOperationName* method also includes an `IAsyncResult` parameter that should be set to the instance returned by the corresponding call to the *BeginOperationName* method.

The Task Parallel Library (TPL) can simplify the process of consuming an APM begin/end method pair by

encapsulating the asynchronous operations in the same `Task` object. This encapsulation is provided by multiple overloads of the `TaskFactory.FromAsync` method.

For more information about APM see [Asynchronous Programming Model](#) and [TPL and Traditional .NET Framework Asynchronous Programming](#) on MSDN.

### Creating the TodoService Object

The generated proxy class provides the `TodoService` class, which is used to communicate with the ASMX service over HTTP. It provides functionality for invoking web service methods as asynchronous operations from a URI identified service instance. For more information about asynchronous operations, see [Async Support Overview](#).

The `TodoService` instance is declared at the class-level so that the object lives for as long as the application needs to consume the ASMX service, as shown in the following code example:

```
public class SoapService : ISoapService
{
    ASMXService.TodoService asmxService;
    ...

    public SoapService ()
    {
        asmxService = new ASMXService.TodoService (Constants.SapUrl);
    }
    ...
}
```

The `TodoService` constructor takes an optional string parameter that specifies the URL of the ASMX service instance. This enables the application to connect to different instances of the ASMX service, provided that there are multiple published instances.

### Creating Data Transfer Objects

The sample application uses the `TodoItem` class to model data. To store a `TodoItem` item in the web service it must first be converted to the proxy generated `TodoItem` type. This is accomplished by the `ToASMXServiceTodoItem` method, as shown in the following code example:

```
ASMXService.TodoItem ToASMXServiceTodoItem (TodoItem item)
{
    return new ASMXService.TodoItem {
        ID = item.ID,
        Name = item.Name,
        Notes = item.Notes,
        Done = item.Done
    };
}
```

This method simply creates a new `ASMXService.TodoItem` instance, and sets each property to the identical property from the `TodoItem` instance.

Similarly, when data is retrieved from the web service, it must be converted from the proxy generated `TodoItem` type to a `TodoItem` instance. This is accomplished with the `FromASMXServiceTodoItem` method, as shown in the following code example:

```

static TodoItem FromASMXServiceTodoItem (ASMXService.TodoItem item)
{
    return new TodoItem {
        ID = item.ID,
        Name = item.Name,
        Notes = item.Notes,
        Done = item.Done
    };
}

```

This method simply retrieves the data from the proxy generated `TodoItem` type and sets it in the newly created `TodoItem` instance.

## Retrieving Data

The `TodoService.BeginGetTodoItems` and `TodoService.EndGetTodoItems` methods are used to call the `GetTodoItems` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```

public async Task<List<TodoItem>> RefreshDataAsync ()
{
    ...
    var todoItems = await Task.Factory.FromAsync<ASMXService.TodoItem[]> (
        todoService.BeginGetTodoItems,
        todoService.EndGetTodoItems,
        null,
        TaskCreationOptions.None);

    foreach (var item in todoItems) {
        Items.Add (FromASMXServiceTodoItem (item));
    }
    ...
}

```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoService.EndGetTodoItems` method once the `TodoService.BeginGetTodoItems` method completes, with the `null` parameter indicating that no data is being passed into the `BeginGetTodoItems` delegate. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The `TodoService.EndGetTodoItems` method returns an array of `ASMXService.TodoItem` instances, which is then converted to a `List` of `TodoItem` instances for display.

## Creating Data

The `TodoService.BeginCreateTodoItem` and `TodoService.EndCreateTodoItem` methods are used to call the `CreateTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```

public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
    ...
    var todoItem = ToASMXServiceTodoItem (item);
    ...
    await Task.Factory.FromAsync (
        todoService.BeginCreateTodoItem,
        todoService.EndCreateTodoItem,
        todoItem,
        TaskCreationOptions.None);
    ...
}

```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoService.EndCreateTodoItem` method once the `TodoService.BeginCreateTodoItem` method completes, with the `todoItem` parameter being the data that's passed into the `BeginCreateTodoItem` delegate to specify the `TodoItem` to be created by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `SoapException` if it fails to create the `TodoItem`, which is handled by the application.

## Updating Data

The `TodoService.BeginEditTodoItem` and `TodoService.EndEditTodoItem` methods are used to call the `EditTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
    ...
    var todoItem = ToASMXServiceTodoItem (item);
    ...
    await Task.Factory.FromAsync (
        todoService.BeginEditTodoItem,
        todoService.EndEditTodoItem,
        todoItem,
        TaskCreationOptions.None);
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoService.EndEditTodoItem` method once the `TodoService.BeginEditTodoItem` method completes, with the `todoItem` parameter being the data that's passed into the `BeginEditTodoItem` delegate to specify the `TodoItem` to be updated by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `SoapException` if it fails to locate or update the `TodoItem`, which is handled by the application.

## Deleting Data

The `TodoService.BeginDeleteTodoItem` and `TodoService.EndDeleteTodoItem` methods are used to call the `DeleteTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```
public async Task DeleteTodoItemAsync (string id)
{
    ...
    await Task.Factory.FromAsync (
        todoService.BeginDeleteTodoItem,
        todoService.EndDeleteTodoItem,
        id,
        TaskCreationOptions.None);
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoService.EndDeleteTodoItem` method once the `TodoService.BeginDeleteTodoItem` method completes, with the `id` parameter being the data that's passed into the `BeginDeleteTodoItem` delegate to specify the `TodoItem` to be deleted by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `SoapException` if it fails to locate or delete the `TodoItem`, which is handled by the application.

## Summary

This article demonstrated how to consume an ASMX web service from a Xamarin.Forms application. ASMX provides the ability to build web services that send messages over HTTP using SOAP. Consumers of an ASMX service do not need to know anything about the platform, object model, or programming language used to implement the service. They only need to understand how to send and receive SOAP messages.

## Related Links

- [TodoASMX \(sample\)](#)
- [IAsyncResult](#)

# Consuming a Windows Communication Foundation (WCF) Web Service

6/8/2018 • 9 minutes to read • [Edit Online](#)

*WCF is Microsoft's unified framework for building service-oriented applications. It enables developers to build secure, reliable, transacted, and interoperable distributed applications. This article demonstrates how to consume an WCF Simple Object Access Protocol (SOAP) service from a Xamarin.Forms application.*

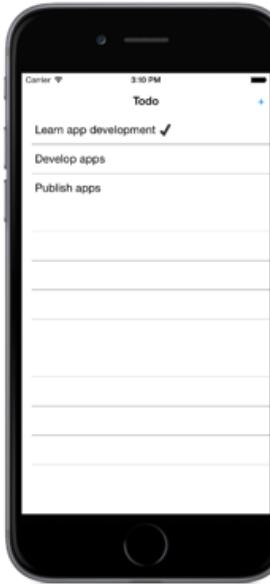
WCF describes a service with a variety of different contracts which include the following:

- **Data contracts** – define the data structures that form the basis for the content within a message.
- **Message contracts** – compose messages from existing data contracts.
- **Fault contracts** – allow custom SOAP faults to be specified.
- **Service contracts** – specify the operations that services support and the messages required for interacting with each operation. They also specify any custom fault behavior that can be associated with operations on each service.

There are differences between ASP.NET Web Services (ASMX) and WCF, but it is important to understand that WCF supports the same capabilities that ASMX provides – SOAP messages over HTTP. For more information about consuming an ASMX service, see [Consuming ASP.NET Web Services \(ASMX\)](#).

In general, the Xamarin platform supports the same client-side subset of WCF that ships with the Silverlight runtime. This includes the most common encoding and protocol implementations of WCF — text-encoded SOAP messages over the HTTP transport protocol using the `BasicHttpBinding` class. In addition, WCF support requires the use of tools only available in a Windows environment to generate the proxy.

Instructions on setting up the WCF service can be found in the readme file that accompanies the sample application. However, when the sample application is run it will connect to a Xamarin-hosted WCF service that provides read-only access to data, as shown in the following screenshot:



The screenshot shows a Windows Phone displaying a todo list application. The screen has a header with the title 'Todo' and a timestamp '3:10 PM'. Below the header is a list of items: 'Learn app development ✓', 'Develop apps', and 'Publish apps'. There is also a large, empty list area for todo items.

```
TodoService.svc.cs
TodoWCFService
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using TodoWCFService.Models;
using TodoWCFService.Services;

namespace TodoWCFService
{
    public class TodoService : ITodoService
    {
        private readonly TodoRepository repository = new TodoRepository();

        public void AddTodoItem(TodoItem item)
        {
            if (!string.IsNullOrEmpty(item.Text) && string.IsNullOrEmpty(item.ID))
                throw new FaultException("ID notes fields are required");

            item.ID = Guid.NewGuid().ToString();
            repository.Add(item);
        }

        public void UpdateTodoItem(TodoItem item)
        {
            if (item.ID == null)
                throw new FaultException("TodoItem ID is in use");
        }
    }
}
```

#### NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

## Consuming the Web Service

The WCF service provides the following operations:

OPERATION	DESCRIPTION	PARAMETERS
GetTodosItems	Get a list of to-do items	
CreateTodoItem	Create a new to-do item	An XML serialized TodoItem
EditTodoItem	Update a to-do item	An XML serialized TodoItem
DeleteTodoItem	Delete a to-do item	An XML serialized TodoItem

For more information about the data model used in the application, see [Modeling the data](#).

#### NOTE

The sample application consumes the Xamarin-hosted WCF service that provides read-only access to the web service. Therefore, the operations that create, update, and delete data will not alter the data consumed in the application. However, a hostable version of the ASMX service is available in the **TodoWCFService** folder in the accompanying sample application. This hostable version of the WCF service permits full create, update, read, and delete access to the data.

A proxy must be generated to consume a WCF service, which allows the application to connect to the service. The proxy is constructed by consuming service metadata that define the methods and associated service configuration. This metadata is exposed in the form of a Web Services Description Language (WSDL) document that is generated by the web service. The proxy can be built by using the Microsoft WCF Web Service Reference Provider in Visual Studio 2017 to add a service reference for the web service to a .NET Standard library. An alternative to creating the proxy using the Microsoft WCF Web Service Reference Provider in Visual Studio 2017 is to use the ServiceModel Metadata Utility Tool (svcutil.exe). For more information, see [ServiceModel Metadata Utility Tool \(Svcutil.exe\)](#).

The generated proxy classes provide methods for consuming the web services that use the Asynchronous Programming Model (APM) design pattern. In this pattern, an asynchronous operation is implemented as two methods named *BeginOperationName* and *EndOperationName*, which begin and end the asynchronous operation.

The *BeginOperationName* method begins the asynchronous operation and returns an object that implements the `IAsyncResult` interface. After calling *BeginOperationName*, an application can continue executing instructions on the calling thread, while the asynchronous operation takes place on a thread pool thread.

For each call to *BeginOperationName*, the application should also call *EndOperationName* to get the results of the operation. The return value of *EndOperationName* is the same type returned by the synchronous web service method. For example, the `EndGetTodoItems` method returns a collection of `TodoItem` instances. The *EndOperationName* method also includes an `IAsyncResult` parameter that should be set to the instance returned by the corresponding call to the *BeginOperationName* method.

The Task Parallel Library (TPL) can simplify the process of consuming an APM begin/end method pair by encapsulating the asynchronous operations in the same `Task` object. This encapsulation is provided by multiple overloads of the `TaskFactory.FromAsync` method.

For more information about APM see [Asynchronous Programming Model](#) and [TPL and Traditional .NET Framework Asynchronous Programming](#) on MSDN.

#### Creating the `TodoServiceClient` Object

The generated proxy class provides the `TodoServiceClient` class, which is used to communicate with the WCF service over HTTP. It provides functionality for invoking web service methods as asynchronous operations from a URI identified service instance. For more information about asynchronous operations, see [Async Support Overview](#).

The `TodoServiceClient` instance is declared at the class-level so that the object lives for as long as the application needs to consume the WCF service, as shown in the following code example:

```

public class SoapService : ISoapService
{
    ITodoService todoService;
    ...

    public SoapService ()
    {
        todoService = new TodoServiceClient (
            new BasicHttpBinding (),
            new EndpointAddress (Constants.SapUrl));
    }
    ...
}

```

The `TodoServiceClient` instance is configured with binding information and an endpoint address. A binding is used to specify the transport, encoding, and protocol details required for applications and services to communicate with each other. The `BasicHttpBinding` specifies that text-encoded SOAP messages will be sent over the HTTP transport protocol. Specifying an endpoint address enables the application to connect to different instances of the WCF service, provided that there are multiple published instances.

For more information about configuring the service reference, see [Configuring the Service Reference](#).

### Creating Data Transfer Objects

The sample application uses the `TodoItem` class to model data. To store a `TodoItem` item in the web service it must first be converted to the proxy generated `TodoItem` type. This is accomplished by the `ToWCFServiceTodoItem` method, as shown in the following code example:

```

TodoWCFService.TodoItem ToWCFServiceTodoItem (TodoItem item)
{
    return new TodoWCFService.TodoItem {
        ID = item.ID,
        Name = item.Name,
        Notes = item.Notes,
        Done = item.Done
    };
}

```

This method simply creates a new `TodoWCFService.TodoItem` instance, and sets each property to the identical property from the `TodoItem` instance.

Similarly, when data is retrieved from the web service, it must be converted from the proxy generated `TodoItem` type to a `TodoItem` instance. This is accomplished with the `FromWCFServiceTodoItem` method, as shown in the following code example:

```

static TodoItem FromWCFServiceTodoItem (TodoWCFService.TodoItem item)
{
    return new TodoItem {
        ID = item.ID,
        Name = item.Name,
        Notes = item.Notes,
        Done = item.Done
    };
}

```

This method simply retrieves the data from the proxy generated `TodoItem` type and sets it in the newly created `TodoItem` instance.

## Retrieving Data

The `TodoServiceClient.BeginGetTodoItems` and `TodoServiceClient.EndGetTodoItems` methods are used to call the `GetTodoItems` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
    ...
    var todoItems = await Task.Factory.FromAsync<ObservableCollection<TodoWCFService.TodoItem>>(
        todoService.BeginGetTodoItems,
        todoService.EndGetTodoItems,
        null,
        TaskCreationOptions.None);

    foreach (var item in todoItems) {
        Items.Add(FromWCFServiceTodoItem(item));
    }
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoServiceClient.EndGetTodoItems` method once the `TodoServiceClient.BeginGetTodoItems` method completes, with the `null` parameter indicating that no data is being passed into the `BeginGetTodoItems` delegate. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The `TodoServiceClient.EndGetTodoItems` method returns an `ObservableCollection` of `TodoWCFService.TodoItem` instances, which is then converted to a `List` of `TodoItem` instances for display.

## Creating Data

The `TodoServiceClient.BeginCreateTodoItem` and `TodoServiceClient.EndCreateTodoItem` methods are used to call the `CreateTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
    ...
    var todoItem = ToWCFServiceTodoItem(item);
    ...

    await Task.Factory.FromAsync(
        todoService.BeginCreateTodoItem,
        todoService.EndCreateTodoItem,
        todoItem,
        TaskCreationOptions.None);
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoServiceClient.EndCreateTodoItem` method once the `TodoServiceClient.BeginCreateTodoItem` method completes, with the `todoItem` parameter being the data that's passed into the `BeginCreateTodoItem` delegate to specify the `TodoItem` to be created by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `FaultException` if it fails to create the `TodoItem`, which is handled by the application.

## Updating Data

The `TodoServiceClient.BeginEditTodoItem` and `TodoServiceClient.EndEditTodoItem` methods are used to call the `EditTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task`

object, as shown in the following code example:

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
    ...
    var todoItem = ToWCFServiceTodoItem (item);
    ...
    await Task.Factory.FromAsync (
        todoService.BeginEditTodoItem,
        todoService.EndEditTodoItem,
        todoItem,
        TaskCreationOptions.None);
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoServiceClient.EndEditTodoItem` method once the `TodoServiceClient.BeginCreateTodoItem` method completes, with the `todoItem` parameter being the data that's passed into the `BeginEditTodoItem` delegate to specify the `TodoItem` to be updated by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `FaultException` if it fails to locate or update the `TodoItem`, which is handled by the application.

## Deleting Data

The `TodoServiceClient.BeginDeleteTodoItem` and `TodoServiceClient.EndDeleteTodoItem` methods are used to call the `DeleteTodoItem` operation provided by the web service. These asynchronous methods are encapsulated in a `Task` object, as shown in the following code example:

```
public async Task DeleteTodoItemAsync (string id)
{
    ...
    await Task.Factory.FromAsync (
        todoService.BeginDeleteTodoItem,
        todoService.EndDeleteTodoItem,
        id,
        TaskCreationOptions.None);
    ...
}
```

The `Task.Factory.FromAsync` method creates a `Task` that executes the `TodoServiceClient.EndDeleteTodoItem` method once the `TodoServiceClient.BeginDeleteTodoItem` method completes, with the `id` parameter being the data that's passed into the `BeginDeleteTodoItem` delegate to specify the `TodoItem` to be deleted by the web service. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

The web service throws a `FaultException` if it fails to locate or delete the `TodoItem`, which is handled by the application.

## Summary

This article demonstrated how to consume an WCF SOAP service from a Xamarin.Forms application. In general, the Xamarin platform supports the same client-side subset of WCF that ships with the Silverlight runtime. This includes the most common encoding and protocol implementations of WCF — text-encoded SOAP messages over the HTTP transport protocol using the `BasicHttpBinding` class. In addition, WCF support requires the use of tools only available in a Windows environment to generate the proxy.

## Related Links

- [TodoWCF \(sample\)](#)
- [IAsyncResult](#)

# Consuming a RESTful Web Service

6/7/2018 • 8 minutes to read • [Edit Online](#)

*Integrating a web service into an application is a common scenario. This article demonstrates how to consume a RESTful web service from a Xamarin.Forms application.*

Representational State Transfer (REST) is an architectural style for building web services. REST requests are made over HTTP using the same HTTP verbs that web browsers use to retrieve web pages and to send data to servers. The verbs are:

- **GET** – this operation is used to retrieve data from the web service.
- **POST** – this operation is used to create a new item of data on the web service.
- **PUT** – this operation is used to update an item of data on the web service.
- **PATCH** – this operation is used to update an item of data on the web service by describing a set of instructions about how the item should be modified. This verb is not used in the sample application.
- **DELETE** – this operation is used to delete an item of data on the web service.

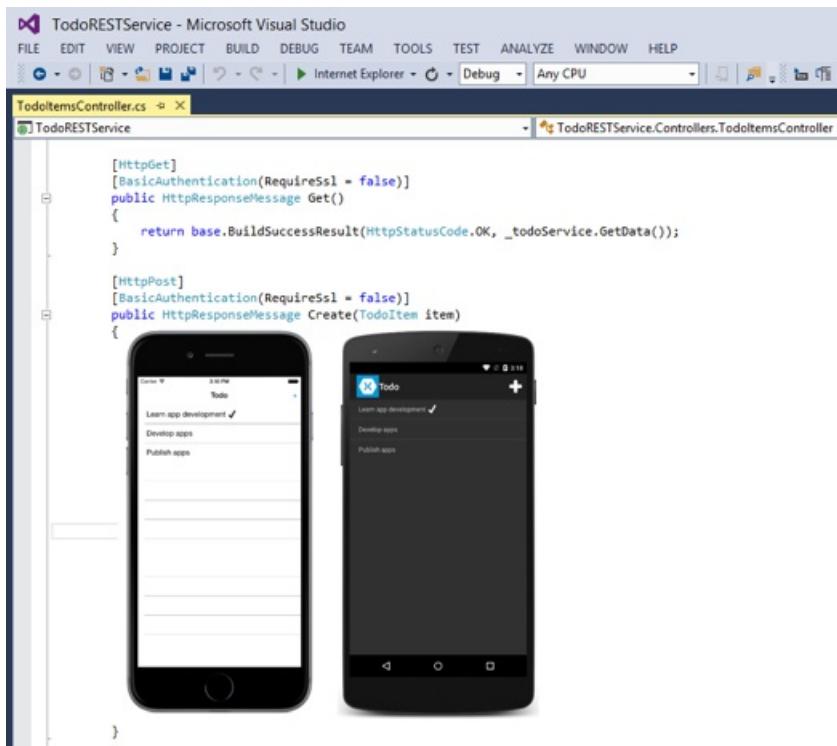
Web service APIs that adhere to REST are called RESTful APIs, and are defined using:

- A base URI.
- HTTP methods, such as GET, POST, PUT, PATCH, or DELETE.
- A media type for the data, such as JavaScript Object Notation (JSON).

RESTful web services typically use JSON messages to return data to the client. JSON is a text-based data-interchange format that produces compact payloads, which results in reduced bandwidth requirements when sending data. The sample application uses the open source [NewtonSoft JSON.NET library](#) to serialize and deserialize messages.

The simplicity of REST has helped make it the primary method for accessing web services in mobile applications.

Instructions on setting up the REST service can be found in the readme file that accompanies the sample application. However, when the sample application is run, it will connect to a Xamarin-hosted REST service that provides read-only access to data, as shown in the following screenshot:



#### NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception.

ATS can be opted out of if it is not possible to use the **HTTPS** protocol and secure communication for internet resources. This can be achieved by updating the app's **Info.plist** file. For more information see [App Transport Security](#).

## Consuming the Web Service

The REST service is written using ASP.NET Core and provides the following operations:

OPERATION	HTTP METHOD	RELATIVE URI	PARAMETERS
Get a list of to-do items	GET	/api/todoitems/	
Create a new to-do item	POST	/api/todoitems/	A JSON formatted TodoItem
Update a to-do item	PUT	/api/todoitems/	A JSON formatted TodoItem
Delete a to-do item	DELETE	/api/todoitems/{id}	

The majority of the URIs include the `TodoItem` ID in the path. For example, to delete the `TodoItem` whose ID is `6bb8a868-dba1-4f1a-93b7-24ebce87e243`, the client sends a DELETE request to `http://hostname/api/todoitems/6bb8a868-dba1-4f1a-93b7-24ebce87e243`. For more information about the data model used in the sample application, see [Modeling the data](#).

When the Web API framework receives a request it routes the request to an action. These actions are simply public methods in the `TodoItemsController` class. The framework uses a routing table to determine which action to invoke in response to a request, which is shown in the following code example:

```
config.Routes.MapHttpRoute(
    name: "TodoItemsApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { controller="todoitems", id = RouteParameter.Optional }
);
```

The routing table contains a route template, and when the Web API framework receives an HTTP request, it tries to match the URL against the route template in the routing table. If a matching route cannot be found the client receives a 404 (not found) error. If a matching route is found, Web API selects the controller and the action as follows:

- To find the controller, Web API adds "controller" to the value of the `{controller}` variable.
- To find the action, Web API looks at the HTTP method and looks at controller actions that are decorated with the same HTTP method as an attribute.
- The `{id}` placeholder variable is mapped to an action parameter.

The REST service uses basic authentication. For more information see [Authenticating a RESTful web service](#). For more information about ASP.NET Web API routing, see [Routing in ASP.NET Web API](#) on the ASP.NET website. For more information about building the REST service using ASP.NET Core, see [Creating Backend Services for Native Mobile Applications](#).

#### NOTE

The sample application consumes the Xamarin-hosted REST service that provides read-only access to the web service. Therefore, the operations that create, update, and delete data will not alter the data consumed in the application. However, a hostable version of the REST service is available in the **TodoRESTService** folder in the accompanying [sample code](#). If you host the REST service yourself, it permits full create, update, read, and delete access to the data.

The `HttpClient` class is used to send and receive requests over HTTP. It provides functionality for sending HTTP requests and receiving HTTP responses from a URI identified resource. Each request is sent as an asynchronous operation. For more information about asynchronous operations, see [Async Support Overview](#).

The `HttpResponseMessage` class represents an HTTP response message received from the web service after an HTTP request has been made. It contains information about the response, including the status code, headers, and any body. The `HttpContent` class represents the HTTP body and content headers, such as `Content-Type` and `Content-Encoding`. The content can be read using any of the `ReadAs` methods, such as `ReadAsStringAsync` and `ReadAsByteArrayAsync`, depending upon the format of the data.

### Creating the HttpClient Object

The `HttpClient` instance is declared at the class-level so that the object lives for as long as the application needs to make HTTP requests, as shown in the following code example:

```
public class RestService : IRestService
{
    HttpClient client;
    ...

    public RestService ()
    {
        client = new HttpClient ();
        client.MaxResponseContentBufferSize = 256000;
    }
    ...
}
```

The `HttpClient.MaxResponseContentBufferSize` property is used to specify the maximum number of bytes to buffer when reading the content in the HTTP response message. The default size of this property is the maximum size of an integer. Therefore, the property is set to a smaller value, as a safeguard, to limit the amount of data that the application will accept as a response from the web service.

## Retrieving Data

The `HttpClient.GetAsync` method is used to send the GET request to the web service specified by the URI, and then receive the response from the web service, as shown in the following code example:

```
public async Task<List<TodoItem>> RefreshDataAsync ()  
{  
    ...  
    // RestUrl = http://developer.xamarin.com:8081/api/todoitems/  
    var uri = new Uri (string.Format (Constants.RestUrl, string.Empty));  
    ...  
    var response = await client.GetAsync (uri);  
    if (response.IsSuccessStatusCode) {  
        var content = await response.Content.ReadAsStringAsync ();  
        Items = JsonConvert.DeserializeObject <List<TodoItem>> (content);  
    }  
    ...  
}
```

The REST service sends an HTTP status code in the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed. For this operation the REST service sends HTTP status code 200 (OK) in the response, which indicates that the request succeeded and that the requested information is in the response.

If the HTTP operation was successful, the content of the response is read, for display. The `HttpResponseMessage.Content` property represents the content of the HTTP response, and the `HttpContent.ReadAsStringAsync` method asynchronously writes the HTTP content to a string. This content is then converted from JSON to a `List` of `TodoItem` instances.

## Creating Data

The `HttpClient.PostAsync` method is used to send the POST request to the web service specified by the URI, and then to receive the response from the web service, as shown in the following code example:

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)  
{  
    // RestUrl = http://developer.xamarin.com:8081/api/todoitems/  
    var uri = new Uri (string.Format (Constants.RestUrl, string.Empty));  
    ...  
    var json = JsonConvert.SerializeObject (item);  
    var content = new StringContent (json, Encoding.UTF8, "application/json");  
  
    HttpResponseMessage response = null;  
    if (isNewItem) {  
        response = await client.PostAsync (uri, content);  
    }  
    ...  
  
    if (response.IsSuccessStatusCode) {  
        Debug.WriteLine (@"                TodoItem successfully saved.");  
    }  
    ...  
}
```

The `TodoItem` instance is converted to a JSON payload for sending to the web service. This payload is then embedded in the body of the HTTP content that will be sent to the web service before the request is made with the `PostAsync` method.

The REST service sends an HTTP status code in the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed. The common responses for this operation are:

- **201 (CREATED)** – the request resulted in a new resource being created before the response was sent.
- **400 (BAD REQUEST)** – the request is not understood by the server.
- **409 (CONFLICT)** – the request could not be carried out because of a conflict on the server.

## Updating Data

The `HttpClient.PutAsync` method is used to send the PUT request to the web service specified by the URI, and then receive the response from the web service, as shown in the following code example:

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
    ...
    response = await client.PutAsync (uri, content);
    ...
}
```

The operation of the `PutAsync` method is identical to the `PostAsync` method that's used for creating data in the web service. However, the possible responses sent from the web service differ.

The REST service sends an HTTP status code in the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed. The common responses for this operation are:

- **204 (NO CONTENT)** – the request has been successfully processed and the response is intentionally blank.
- **400 (BAD REQUEST)** – the request is not understood by the server.
- **404 (NOT FOUND)** – the requested resource does not exist on the server.

## Deleting Data

The `HttpClient.DeleteAsync` method is used to send the DELETE request to the web service specified by the URI, and then receive the response from the web service, as shown in the following code example:

```
public async Task DeleteTodoItemAsync (string id)
{
    // RestUrl = http://developer.xamarin.com:8081/api/todoitems/{0}
    var uri = new Uri (string.Format (Constants.RestUrl, id));
    ...
    var response = await client.DeleteAsync (uri);
    if (response.IsSuccessStatusCode) {
        Debug.WriteLine (@"
            TodoItem successfully deleted.");
    }
    ...
}
```

The REST service sends an HTTP status code in the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed. The common responses for this operation are:

- **204 (NO CONTENT)** – the request has been successfully processed and the response is intentionally blank.
- **400 (BAD REQUEST)** – the request is not understood by the server.
- **404 (NOT FOUND)** – the requested resource does not exist on the server.

## Summary

This article examined how to consume a RESTful web service from a Xamarin.Forms application, using the `HttpClient` class. The simplicity of REST has helped make it the primary method for accessing web services in mobile applications.

## Related Links

- [Creating Backend Services for Native Mobile Applications](#)
- [TodoREST \(sample\)](#)
- [HttpClient](#)

# Consuming an Azure Mobile App

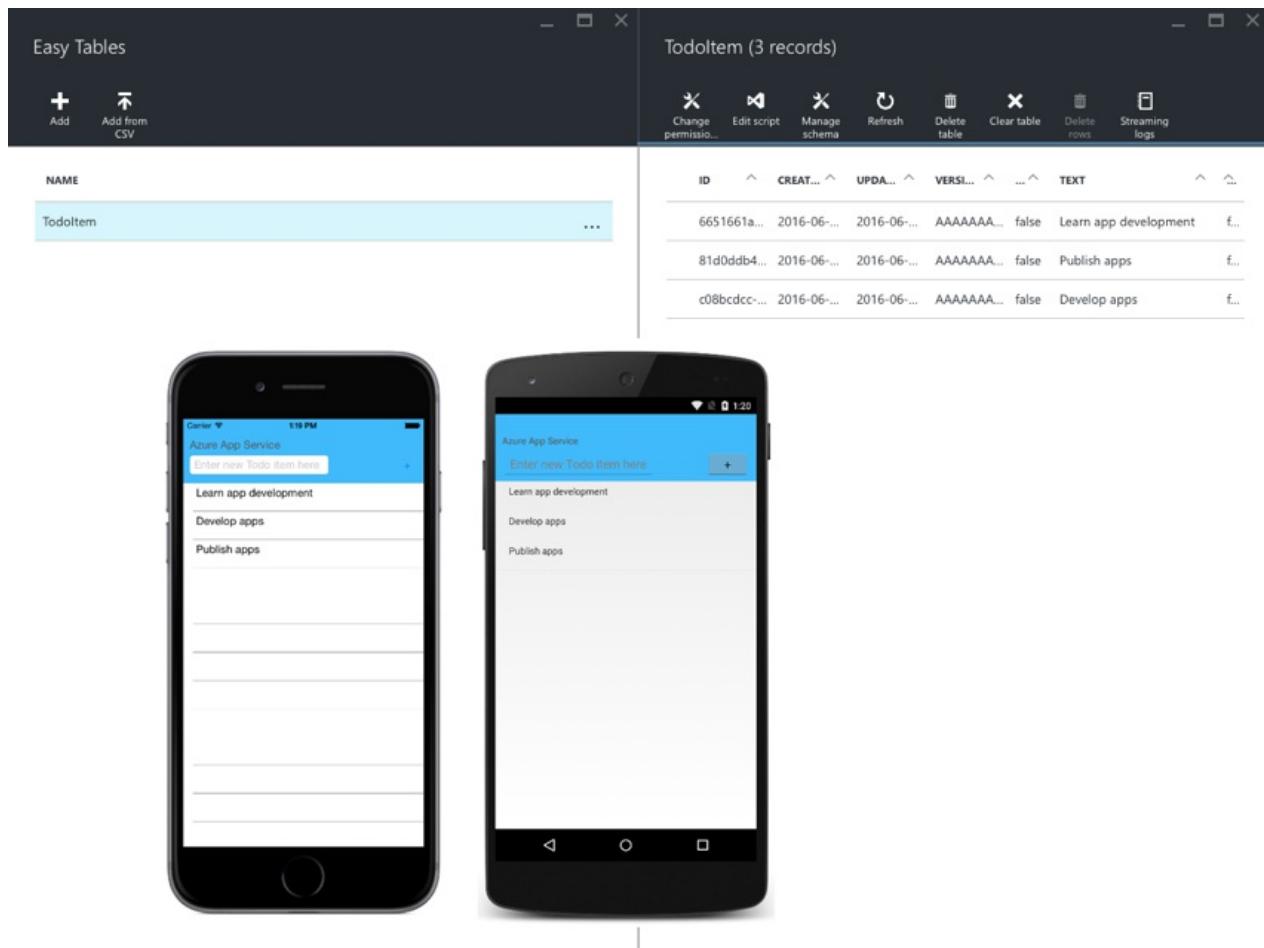
6/20/2018 • 4 minutes to read • [Edit Online](#)

Azure Mobile Apps allow you to develop apps with scalable backends hosted in Azure App Service, with support for mobile authentication, offline sync, and push notifications. This article, which is only applicable to Azure Mobile Apps that use a Node.js backend, explains how to query, insert, update, and delete data stored in a table in an Azure Mobile Apps instance.

## NOTE

Starting on June 30, all new Azure Mobile Apps will be created with TLS 1.2 by default. In addition, it's also recommended that existing Azure Mobile Apps be reconfigured to use TLS 1.2. For information on how to enforce TLS 1.2 in an Azure Mobile App, see [Enforce TLS 1.2](#). For information on how to configure Xamarin projects to use TLS 1.2, see [Transport Layer Security \(TLS\) 1.2](#).

For information on how to create an Azure Mobile Apps instance that can be consumed by Xamarin.Forms, see [Create a Xamarin.Forms app](#). After following these instructions, the downloadable sample application can be configured to consume the Azure Mobile Apps instance by setting the `Constants.ApplicationURL` to the URL of the Azure Mobile Apps instance. Then, when the sample application is run it will connect to the Azure Mobile Apps instance, as shown in the following screenshot:



Access to Azure Mobile Apps is through the [Azure Mobile Client SDK](#), and all connections from the Xamarin.Forms sample application to Azure are made over HTTPS.

#### NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

## Consuming an Azure Mobile App Instance

The [Azure Mobile Client SDK](#) provides the `MobileServiceClient` class, which is used by a Xamarin.Forms application to access the Azure Mobile Apps instance, as shown in the following code example:

```
IMobileServiceTable<TodoItem> todoTable;
MobileServiceClient client;

public TodoItemManager ()
{
    client = new MobileServiceClient (Constants.ApplicationURL);
    todoTable = client.GetTable<TodoItem> ();
}
```

When the `MobileServiceClient` instance is created, an application URL must be specified to identify the Azure Mobile Apps instance. This value can be obtained from the dashboard for the mobile app in the [Microsoft Azure Portal](#).

A reference to the `TodoItem` table stored in the Azure Mobile Apps instance must be obtained before operations can be performed on that table. This is achieved by calling the `GetTable` method on the `MobileServiceClient` instance, which returns a `IMobileServiceTable<TodoItem>` reference.

### Querying Data

The contents of a table can be retrieved by calling the `IMobileServiceTable.ToEnumerableAsync` method that asynchronously evaluates the query and returns the results. Data can also be filtered server side by including a `Where` clause in the query. The `Where` clause applies a row filtering predicate to the query against the table, as shown in the following code example:

```
public async Task<ObservableCollection<TodoItem>> GetTodoItemsAsync (bool syncItems = false)
{
    ...
    IEnumerable<TodoItem> items = await todoTable
        .Where (todoItem => !todoItem.Done)
        .ToEnumerableAsync ();

    return new ObservableCollection<TodoItem> (items);
}
```

This query returns all the items from the `TodoItem` table whose `Done` property is equal to `false`. The query results are then placed in an `ObservableCollection` for display.

### Inserting Data

When inserting data in the Azure Mobile Apps instance, new columns will automatically be generated in the table as required, provided that dynamic schema is enabled in the Azure Mobile Apps instance. The `IMobileServiceTable.InsertAsync` method is used to insert a new row of data into the specified table, as shown in the following code example:

```
public async Task SaveTaskAsync (TodoItem item)
{
    ...
    await todoTable.InsertAsync (item);
    ...
}
```

When making an insert request, an ID must not be specified in the data being passed to the Azure Mobile Apps instance. If the insert request contains an ID a `MobileServiceInvalidOperationException` will be thrown.

After the `InsertAsync` method completes, the ID of the data in the Azure Mobile Apps instance will be populated in the `TodoItem` instance in the Xamarin.Forms application.

## Updating Data

When updating data in the Azure Mobile Apps instance, new columns will automatically be generated in the table as required, provided that dynamic schema is enabled in the Azure Mobile Apps instance. The `IMobileServiceTable.UpdateAsync` method is used to update existing data with new information, as shown in the following code example:

```
public async Task SaveTaskAsync (TodoItem item)
{
    ...
    await todoTable.UpdateAsync (item);
    ...
}
```

When making an update request, an ID must be specified so that the Azure Mobile Apps instance can identify the data to be updated. This ID value is stored in the `TodoItem.ID` property. If the update request doesn't contain an ID there is no way for the Azure Mobile Apps instance to determine the data to be updated, and so a `MobileServiceInvalidOperationException` will be thrown.

## Deleting Data

The `IMobileServiceTable.DeleteAsync` method is used to delete data from an Azure Mobile Apps table, as shown in the following code example:

```
public async Task DeleteTaskAsync (TodoItem item)
{
    ...
    await todoTable.DeleteAsync(item);
    ...
}
```

When making a delete request, an ID must be specified so that the Azure Mobile App sinstance can identify the data to be deleted. This ID value is stored in the `TodoItem.ID` property. If the delete request doesn't contain an ID, there is no way for the Azure Mobile Apps instance to determine the data to be deleted, and so a `MobileServiceInvalidOperationException` will be thrown.

## Summary

This article explained how to use the [Azure Mobile Client SDK](#) to query, insert, update, and delete data stored in a table in an Azure Mobile apps instance. The SDK provides the `MobileServiceClient` class that is used by a Xamarin.Forms application to access the Azure Mobile Apps instance.

## Related Links

- [TodoAzure \(sample\)](#)
- [Create a Xamarin.Forms app](#)
- [Azure Mobile Client SDK](#)
- [MobileServiceClient](#)

# Authenticating Access to Web Services

6/8/2018 • 2 minutes to read • [Edit Online](#)

This guide explains how to integrate authentication services into a Xamarin.Forms application to enable users to share a backend while only having access to their own data. Topics covered include using basic authentication with a REST service, using the Xamarin.Auth component to authenticate against OAuth identity providers, and using the built-in authentication mechanisms offered by different providers.

## Authenticating a RESTful Web Service

HTTP supports the use of several authentication mechanisms to control access to resources. Basic authentication provides access to resources to only those clients that have the correct credentials. This article demonstrates how to use basic authentication to protect access to RESTful web service resources.

## Authenticating Users with an Identity Provider

Xamarin.Auth is a cross-platform SDK for authenticating users and storing their accounts. It includes OAuth authenticators that provide support for consuming identity providers such as Google, Microsoft, Facebook, and Twitter. This article explains how to use Xamarin.Auth to manage the authentication process in a Xamarin.Forms application.

## Authenticating Users with Azure Mobile Apps

Azure Mobile Apps use a variety of external identity providers to support authenticating and authorizing application users. Permissions can then be set on tables to restrict access to only authenticated users. This article explains how to use Azure Mobile Apps to manage the authentication process in a Xamarin.Forms application.

## Authenticating Users with Azure Active Directory B2C

Azure Active Directory B2C is a cloud identity management solution for consumer-facing web and mobile applications. This article demonstrates how to use Microsoft Authentication Library (MSAL) and Azure Active Directory B2C to integrate consumer identity management into a Xamarin.Forms application.

## Integrating Azure Active Directory B2C with Azure Mobile Apps

Azure Active Directory B2C can be used to manage the authentication workflow for Azure Mobile Apps. With this approach, the identity management experience is fully defined in the cloud, and can be modified without changing your mobile application code. This article demonstrates how to use Azure Active Directory B2C to provide authentication and authorization to an Azure Mobile Apps instance with Xamarin.Forms.

## Related Links

- [Introduction to Web Services](#)
- [Async Support Overview](#)

# Authenticating a RESTful Web Service

6/8/2018 • 4 minutes to read • [Edit Online](#)

HTTP supports the use of several authentication mechanisms to control access to resources. Basic authentication provides access to resources to only those clients that have the correct credentials. This article demonstrates how to use basic authentication to protect access to RESTful web service resources.

The accompanying Xamarin.Forms sample application consumes a Xamarin-hosted REST service that provides read-only access to the web service. Therefore, the operations that create, update, and delete data will not alter the data consumed in the application. However, a hostable version of the REST service is available in the *TodoRESTService* folder in the sample application, and instructions on setting up the service can be found there. This hostable version of the REST service provides full create, update, read, and delete access to the data.

## NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

## Authenticating Users over HTTP

Basic authentication is the simplest authentication mechanism supported by HTTP, and involves the client sending the username and password as unencrypted base64 encoded text. It works as follows:

- If a web service receives a request for a protected resource, it rejects the request with an HTTP status code 401 (access denied) and sets the `WWW-Authenticate` response header, as shown in the following diagram:



- If a web service receives a request for a protected resource, with the `Authorization` header correctly set, the web service responds with an HTTP status code 200, which indicates that the request succeeded and that the requested information is in the response. This scenario is shown in the following diagram:



#### NOTE

Basic authentication should only be used over an HTTPS connection. When used over an HTTP connection, the `Authorization` header can easily be decoded if the HTTP traffic is captured by an attacker.

## Specifying Basic Authentication in a Web Request

Use of basic authentication is specified as follows:

1. The string "Basic " is added to the `Authorization` header of the request.
2. The username and password are combined into a string with the format "username:password", which is then base64 encoded and added to the `Authorization` header of the request.

Therefore, with a username of 'XamarinUser' and a password of 'XamarinPassword', the header becomes:

```
Authorization: Basic WG FtYXJpb lVzZXi6WG FtYXJpb lBhc3N3b3Jk
```

The `HttpClient` class can set the `Authorization` header value on the `HttpClient.DefaultRequestHeaders.Authorization` property. Because the `HttpClient` instance exists across multiple requests, the `Authorization` header needs only to be set once, rather than when making every request, as shown in the following code example:

```
public class RestService : IRestService
{
    HttpClient client;
    ...

    public RestService ()
    {
        var authData = string.Format ("{0}:{1}", Constants.Username, Constants.Password);
        var authHeaderValue = Convert.ToBase64String (Encoding.UTF8.GetBytes (authData));

        client = new HttpClient ();
        ...
        client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue ("Basic", authHeaderValue);
    }
    ...
}
```

Then when a request is made to a web service operation the request is signed with the `Authorization` header, indicating whether or not the user has permission to invoke the operation.

#### NOTE

While the sample REST service stores credentials as constants, they should not be stored in an insecure format in a published application. The [Xamarith.Auth](#) NuGet provides functionality for securely storing credentials. For more information see [Storing and retrieving account information on devices](#).

## Processing the Authorization Header Server Side

The accompanying sample REST service decorates each action with the `[BasicAuthentication]` attribute. This attribute is implemented by the `BasicAuthenticationAttribute` class in the solution, and is used to parse the `Authorization` header and determine if the base64 encoded credentials are valid by comparing them against values stored in `Web.config`. While this approach is suitable for the sample service, it requires extending for a

public-facing web service.

In the basic authentication module used by IIS, users are authenticated against their Windows credentials. Therefore, users must have accounts on the server's domain. However, the Basic authentication model can be configured to allow custom authentication, where user accounts are authenticated against an external source, such as a database. For more information see [Basic Authentication in ASP.NET Web API](#) on the ASP.NET website.

**NOTE**

Basic authentication was not designed to manage logging out. Therefore, the standard basic authentication approach for logging out is to end the session.

## Summary

This article demonstrated how to add basic authentication to web requests made by a Xamarin.Forms application using the `HttpClient` class. Basic authentication provides access to resources to only those clients that have the correct credentials. For information about how to use [Xamarin.Auth](#) to manage the authentication process in a Xamarin.Forms application so that users can share a backend while only having access to their data, see [Authenticating Users with an Identity Provider](#).

## Related Links

- [TodoREST \(sample\)](#)
- [Consuming a RESTful web service](#)
- [HttpClient](#)

# Authenticating Users with an Identity Provider

7/5/2018 • 12 minutes to read • [Edit Online](#)

*Xamarin.Auth is a cross-platform SDK for authenticating users and storing their accounts. It includes OAuth authenticators that provide support for consuming identity providers such as Google, Microsoft, Facebook, and Twitter. This article explains how to use Xamarin.Auth to manage the authentication process in a Xamarin.Forms application.*

OAuth is an open standard for authentication, and enables a resource owner to notify a resource provider that permission should be granted to a third party to access their information without sharing the resource owners identity. An example of this would be enabling a user to notify an identity provider (such as Google, Microsoft, Facebook, or Twitter) that permission should be granted to an application to access their data, without sharing the user's identity. It is commonly used as an approach for users to sign-in to websites and applications using an identity provider, but without exposing their password to the website or application.

A high-level overview of the authentication flow when consuming an OAuth identity provider is as follows:

1. The application navigates a browser to an identity provider URL.
2. The identity provider handles user authentication and returns an authorization code to the application.
3. The application exchanges the authorization code for an access token from the identity provider.
4. The application uses the access token to access APIs on the identity provider, such as an API for requesting basic user data.

The sample application demonstrates how to use Xamarin.Auth to implement a native authentication flow against Google. While Google is used as the identity provider in this topic, the approach is equally applicable to other identity providers. For more information about authentication using Google's OAuth 2.0 endpoint, see [Using OAuth2.0 to Access Google APIs](#) on Google's website.

## NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

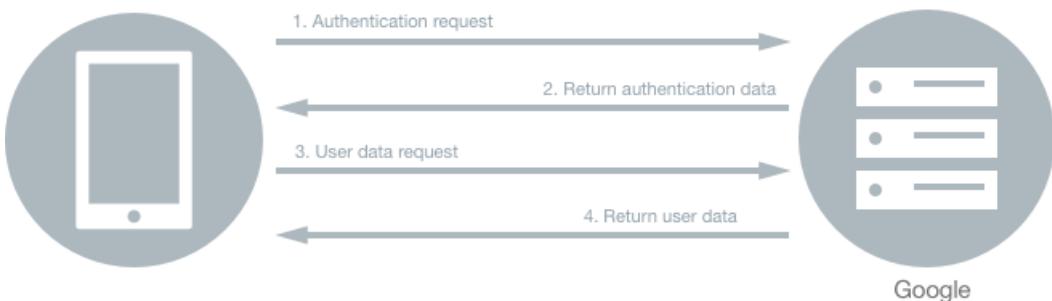
## Using Xamarin.Auth to Authenticate Users

Xamarin.Auth supports two approaches for applications to interact with an identity provider's authorization endpoint:

1. Using an embedded web view. While this has been a common practice, it's no longer recommended for the following reasons:
  - The application that hosts the web view can access the user's full authentication credential, not just the OAuth authorization grant that was intended for the application. This violates the principle of least privilege, as the application has access to more powerful credentials than it requires, potentially increasing the attack surface of the application.
  - The host application could capture usernames and passwords, automatically submit forms and bypass user-consent, and copy session cookies and use them to perform authenticated actions as the user.

- Embedded web views don't share the authentication state with other applications, or the device's web browser, requiring the user to sign-in for every authorization request which is considered an inferior user experience.
  - Some authorization endpoints take steps to detect and block authorization requests that come from web views.
2. Using the device's web browser, which is the recommended approach. Using the device browser for OAuth requests improves the usability of an application, as users only need to sign-in to the identity provider once per device, improving conversion rates of sign-in and authorization flows in the application. The device browser also provides improved security as applications are able to inspect and modify content in a web view, but not content shown in the browser. This is the approach taken in this article and sample application.

A high-level overview of how the sample application uses Xamarin.Auth to authenticate users and retrieve their basic data is shown in the following diagram:



The application makes an authentication request to Google using the `OAuth2Authenticator` class. An authentication response is returned, once the user has successfully authenticated with Google through their sign-in page, which includes an access token. The application then makes a request to Google for basic user data, using the `OAuth2Request` class, with the access token being included in the request.

## Setup

A Google API Console project must be created to integrate Google sign-in with a Xamarin.Forms application. This can be accomplished as follows:

1. Go to the [Google API Console](#) website, and sign in with Google account credentials.
2. From the project drop-down, select an existing project, or create a new one.
3. In the sidebar under "API Manager", select **Credentials**, then select the **OAuth consent screen tab**. Choose an **Email address**, specify a **Product name shown to users**, and press **Save**.
4. In the **Credentials** tab, select the **Create credentials** drop-down list, and choose **OAuth client ID**.
5. Under **Application type**, select the platform that the mobile application will be running on (**iOS** or **Android**).
6. Fill in the required details and select the **Create** button.

### NOTE

A Client ID lets an application access enabled Google APIs, and for mobile applications is unique to a single platform. Therefore, a **OAuth client ID** should be created for each platform that will use Google sign-in.

After performing these steps, Xamarin.Auth can be used to initiate an OAuth2 authentication flow with Google.

## Creating and Configuring an Authenticator

Xamarin.Auth's `OAuth2Authenticator` class is responsible for handling the OAuth authentication flow. The following code example shows the instantiation of the `OAuth2Authenticator` class when performing authentication using the device's web browser:

```
var authenticator = new OAuth2Authenticator(
    clientId,
    null,
    Constants.Scope,
    new Uri(Constants.AuthorizeUrl),
    new Uri(redirectUri),
    new Uri(Constants.AccessTokenUrl),
    null,
    true);
```

The `OAuth2Authenticator` class requires a number of parameters, which are as follows:

- **Client ID** – this identifies the client that is making the request, and can be retrieved from the project in the [Google API Console](#).
- **Client Secret** – this should be `null` or `string.Empty`.
- **Scope** – this identifies the API access being requested by the application, and the value informs the consent screen that is shown to the user. For more information about scopes, see [Authorizing API request](#) on Google's website.
- **Authorize URL** – this identifies the URL where the authorization code will be obtained from.
- **Redirect URL** – this identifies the URL where the response will be sent. The value of this parameter must match one of the values that appears in the **Credentials** tab for the project in the [Google Developers Console](#).
- **AccessToken Url** – this identifies the URL used to request access tokens after an authorization code is obtained.
- **GetUserNameAsync Func** – an optional `Func` that will be used to asynchronously retrieve the username of the account after it's been successfully authenticated.
- **Use Native UI** – a `boolean` value indicating whether to use the device's web browser to perform the authentication request.

## Setup Authentication Event Handlers

Before presenting the user interface, an event handler for the `OAuth2Authenticator.Completed` event must be registered, as shown in the following code example:

```
authenticator.Completed += OnAuthCompleted;
```

This event will fire when the user successfully authenticates or cancels the sign-in.

Optionally, an event handler for the `OAuth2Authenticator.Error` event can also be registered.

## Presenting the Sign-In User Interface

The sign-in user interface can be presented to the user by using a `Xamarin.Auth` login presenter, which must be initialized in each platform project. The following code example shows how to initialize a login presenter in the `AppDelegate` class in the iOS project:

```
global::Xamarin.Auth.Presenters.XamarinIOS.AuthenticationConfiguration.Init();
```

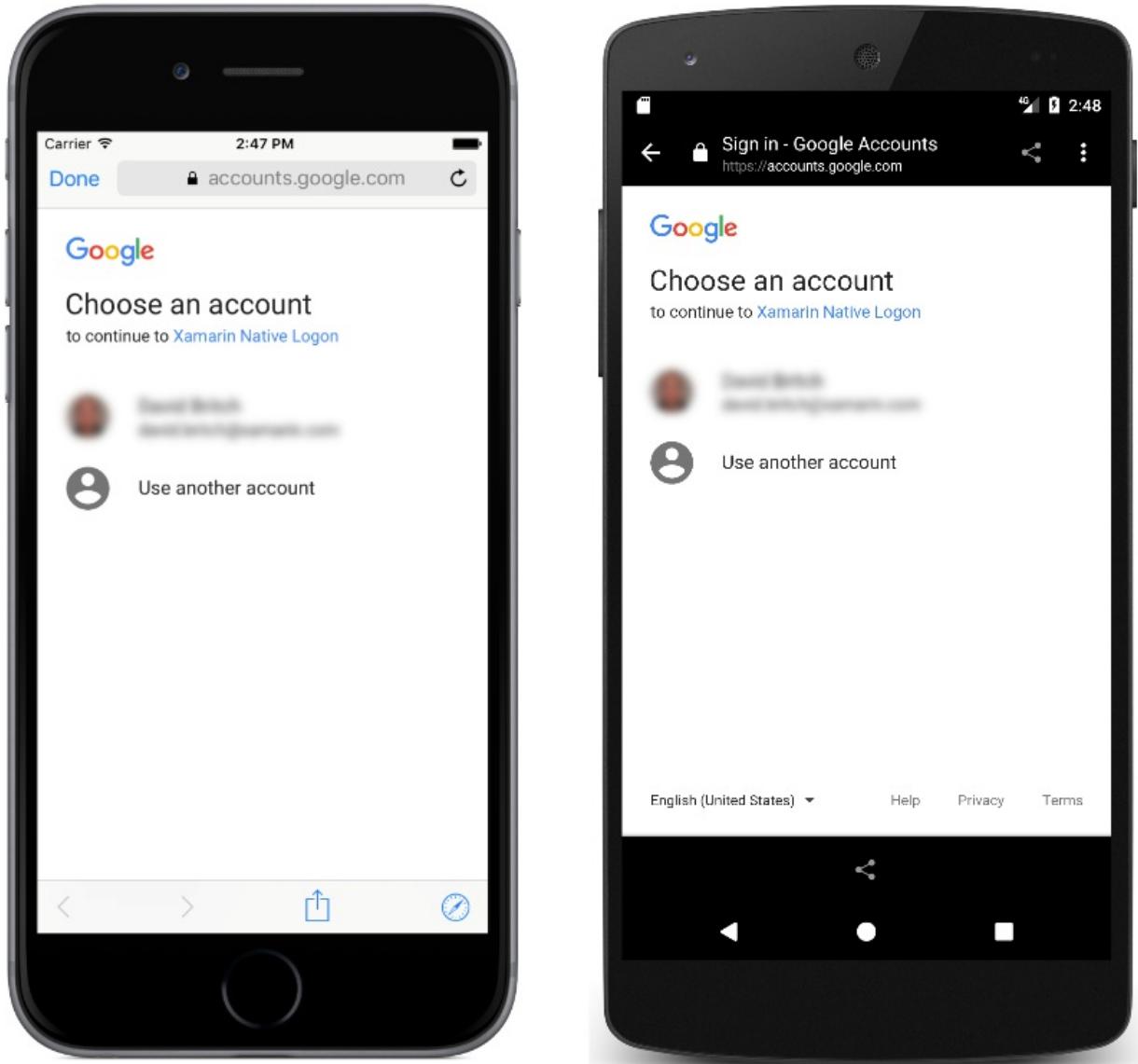
The following code example shows how to initialize a login presenter in the `MainActivity` class in the Android project:

```
global::Xamarin.Auth.Presenters.XamarinAndroid.AuthenticationConfiguration.Init(this, bundle);
```

The .NET Standard library project can then invoke the login presenter as follows:

```
var presenter = new Xamarin.Auth.Presenters.OAuthLoginPresenter();
presenter.Login(authenticator);
```

Note that the argument to the `Xamarin.Auth.Presenters.OAuthLoginPresenter.Login` method is the `OAuth2Authenticator` instance. When the `Login` method is invoked, the sign-in user interface is presented to the user in a tab from the device's web browser, which is shown in the following screenshots:



### Processing the Redirect URL

After the user completes the authentication process, control will return to the application from the web browser tab. This is achieved by registering a custom URL scheme for the redirect URL that's returned from the authentication process, and then detecting and handling the custom URL once it's sent.

When choosing a custom URL scheme to associate with an application, applications must use a scheme based on a name under their control. This can be achieved by using the bundle identifier name on iOS, and the package name on Android, and then reversing them to make the URL scheme. However, some identity providers, such as Google, assign client identifiers based on domain names, which are then reversed and used as the URL scheme. For example, if Google creates a client id of

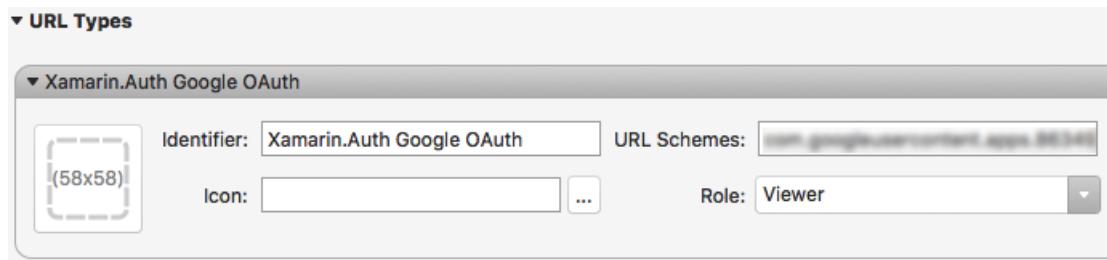
`902730282010-ks3kd03ksoasioda93j1das93jjj22kr.apps.googleusercontent.com`, the URL scheme will be `com.googleusercontent.apps.902730282010-ks3kd03ksoasioda93j1das93jjj22kr`. Note that only a single `/` can appear after the scheme component. Therefore, a complete example of a redirect URL utilizing a custom URL scheme is `com.googleusercontent.apps.902730282010-ks3kd03ksoasioda93j1das93jjj22kr:/oauth2redirect`.

When the web browser receives a response from the identity provider that contains a custom URL scheme, it tries to load the URL, which will fail. Instead, the custom URL scheme is reported to the operating system by raising an event. The operating system then checks for registered schemes, and if one is found, the operating system will launch the application that registered the scheme, and send it the redirect URL.

The mechanism for registering a custom URL scheme with the operating system and handling the scheme is specific to each platform.

#### iOS

On iOS, a custom URL scheme is registered in **Info.plist**, as shown in the following screenshot:



The **Identifier** value can be anything, and the **Role** value must be set to **Viewer**. The **Url Schemes** value, which begins with `com.googleusercontent.apps`, can be obtained from the iOS client id for the project on [Google API Console](#).

When the identity provider completes the authorization request, it redirects to the application's redirect URL. Because the URL uses a custom scheme it results in iOS launching the application, passing in the URL as a launch parameter, where it's processed by the `OpenUrl` override of the application's `AppDelegate` class, which is shown in the following code example:

```
public override bool OpenUrl(UIApplication app, NSUrl url, NSDictionary options)
{
    // Convert NSUrl to Uri
    var uri = new Uri(url.AbsoluteString);

    // Load redirectUrl page
    AuthenticationState.Authenticator.OnPageLoading(uri);

    return true;
}
```

The `OpenUrl` method converts the received URL from an `NSUrl` to a .NET `uri`, before processing the redirect URL with the `OnPageLoading` method of a public `OAuth2Authenticator` object. This causes Xamarin.Auth to close the web browser tab, and to parse the received OAuth data.

#### Android

On Android, a custom URL scheme is registered by specifying an `IntentFilter` attribute on the `Activity` that will handle the scheme. When the identity provider completes the authorization request, it redirects to the application's redirect URL. As the URL uses a custom scheme it results in Android launching the application, passing in the URL as a launch parameter, where it's processed by the `OnCreate` method of the `Activity` registered to handle the custom URL scheme. The following code example shows the class from the sample application that handles the custom URL scheme:

```
[Activity(Label = "CustomUrlSchemeInterceptorActivity", NoHistory = true, LaunchMode = LaunchMode.SingleTop )]
[IntentFilter(
    new[] { Intent.ActionView },
    Categories = new [] { Intent.CategoryDefault, Intent.CategoryBrowsable },
    DataSchemes = new [] { "<insert custom URL here>" },
    DataPath = "/oauth2redirect")]
public class CustomUrlSchemeInterceptorActivity : Activity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);

        // Convert Android.Net.Url to Uri
        var uri = new Uri(Intent.Data.ToString());

        // Load redirectUrl page
        AuthenticationState.Authenticator.OnPageLoading(uri);

        Finish();
    }
}
```

The `DataSchemes` property of the `IntentFilter` must be set to the reversed client identifier that's obtained from the Android client id for the project on [Google API Console](#).

The `OnCreate` method converts the received URL from an `Android.Net.Url` to a .NET `Uri`, before processing the redirect URL with the `OnPageLoading` method of a public `OAuth2Authenticator` object. This causes Xamarin.Auth to close the web browser tab, and parse the received OAuth data.

#### IMPORTANT

On Android, Xamarin.Auth uses the `CustomTabs` API to communicate with the web browser and operating system. However, it's not guaranteed that a `CustomTabs` compatible browser will be installed on the user's device.

## Examining the OAuth Response

After parsing the received OAuth data, Xamarin.Auth will raise the `OAuth2Authenticator.Completed` event. In the event handler for this event, the `AuthenticatorCompletedEventArgs.IsAuthenticated` property can be used to identify whether authentication succeeded, as shown in the following code example:

```
async void OnAuthCompleted(object sender, AuthenticatorCompletedEventArgs e)
{
    ...
    if (e.IsAuthenticated)
    {
        ...
    }
}
```

The data gathered from a successful authentication is available in the `AuthenticatorCompletedEventArgs.Account` property. This includes an access token, which can be used to sign requests for data to an API provided by the identity provider.

## Making Requests for Data

After the application obtains an access token, it's used to make a request to the `https://www.googleapis.com/oauth2/v2/userinfo` API, to request basic user data from the identity provider. This request is made with Xamarin.Auth's `OAuth2Request` class, which represents a request that is authenticated using an account retrieved from an `OAuth2Authenticator` instance, as shown in the following code example:

```
// UserInfoUrl = https://www.googleapis.com/oauth2/v2/userinfo
var request = new OAuth2Request ("GET", new Uri (Constants.UserInfoUrl), null, e.Account);
var response = await request.GetResponseAsync ();
if (response != null)
{
    string userJson = response.GetResponseText ();
    var user = JsonConvert.DeserializeObject<User> (userJson);
}
```

As well as the HTTP method and the API URL, the `OAuth2Request` instance also specifies an `Account` instance that contains the access token that signs the request to the URL specified by the `Constants.UserInfoUrl` property. The identity provider then returns basic user data as a JSON response, including the users' name and email address, provided that it recognizes the access token as being valid. The JSON response is then read and deserialized into the `user` variable.

For more information, see [Calling a Google API](#) on the Google Developers portal.

### Storing and Retrieving Account Information on Devices

Xamarin.Auth securely stores `Account` objects in an account store so that applications do not always have to re-authenticate users. The `AccountStore` class is responsible for storing account information, and is backed by Keychain services in iOS, and the `KeyStore` class in Android.

The following code example shows how an `Account` object is securely saved:

```
AccountStore.Create ().Save (e.Account, Constants.AppName);
```

Saved accounts are uniquely identified using a key composed of the account's `Username` property and a service ID, which is a string that is used when fetching accounts from the account store. If an `Account` was previously saved, calling the `Save` method again will overwrite it.

`Account` objects for a service can be retrieved by calling the `FindAccountsForService` method, as shown in the following code example:

```
var account = AccountStore.Create ().FindAccountsForService (Constants.AppName).FirstOrDefault();
```

The `FindAccountsForService` method returns an `IEnumerable` collection of `Account` objects, with the first item in the collection being set as the matched account.

## Summary

This article explained how to use Xamarin.Auth to manage the authentication process in a Xamarin.Forms application. Xamarin.Auth provides the `OAuth2Authenticator` and `OAuth2Request` classes that are used by Xamarin.Forms applications to consume identity providers such as Google, Microsoft, Facebook, and Twitter.

## Related Links

- [OAuthNativeFlow \(sample\)](#)
- [OAuth 2.0 for Native Apps](#)
- [Using OAuth2.0 to Access Google APIs](#)
- [Xamarin.Auth \(NuGet\)](#)
- [Xamarin.Auth \(GitHub\)](#)

# Authenticating Users with Azure Mobile Apps

6/8/2018 • 6 minutes to read • [Edit Online](#)

Azure Mobile Apps use a variety of external identity providers to support authenticating and authorizing application users, including Facebook, Google, Microsoft, Twitter, and Azure Active Directory. Permissions can be set on tables to restrict access to authenticated users only. This article explains how to use Azure Mobile Apps to manage the authentication process in a Xamarin.Forms application.

## Overview

The process for having Azure Mobile Apps manage the authentication process in a Xamarin.Forms application is as follows:

1. Register your Azure Mobile App at an identity provider's site, and then set the provider-generated credentials in the Mobile Apps back end. For more information, see [Register your app for authentication and configure App Services](#).
2. Define a new URL scheme for your Xamarin.Forms application, which allows the authentication system to redirect back to Xamarin.Forms application once the authentication process is complete. For more information, see [Add your app to the Allowed External Redirect URLs](#).
3. Restrict access to the Azure Mobile Apps back end to only authenticated clients. For more information, see [Restrict permissions to authenticated users](#).
4. Invoke authentication from the Xamarin.Forms application. For more information, see [Add authentication to the portable class library](#), [Add authentication to the iOS app](#), [Add authentication to the Android app](#), and [Add authentication to Windows 10 app projects](#).

### NOTE

In iOS 9 and greater, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception. ATS can be opted out of if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's `Info.plist` file. For more information see [App Transport Security](#).

Historically, mobile applications have used embedded web views to perform authentication with identity provider's. This is no longer recommended for the following reasons:

- The application that hosts the web view can access the user's full authentication credential, not just the authorization grant that was intended for the application. This violates the principle of least privilege, as the application has access to more powerful credentials than it requires, potentially increasing the attack surface of the application.
- The host application could capture usernames and passwords, automatically submit forms and bypass user-consent, and copy session cookies and use them to perform authenticated actions as the user.
- Embedded web views don't share the authentication state with other applications, or the device's web browser, requiring the user to sign-in for every authorization request which is considered an inferior user experience.
- Some authorization endpoints take steps to detect and block authorization requests that come from web views.

The alternative is to use the device's web browser to perform authentication, which is the approach taken by the latest version of the [Azure Mobile Client SDK](#). Using the device browser for authentication requests improves the

usability of an application, as users only need to sign-in to the identity provider once per device, improving conversion rates of sign-in and authorization flows in the application. The device browser also provides improved security as applications are able to inspect and modify content in a web view, but not content shown in the browser.

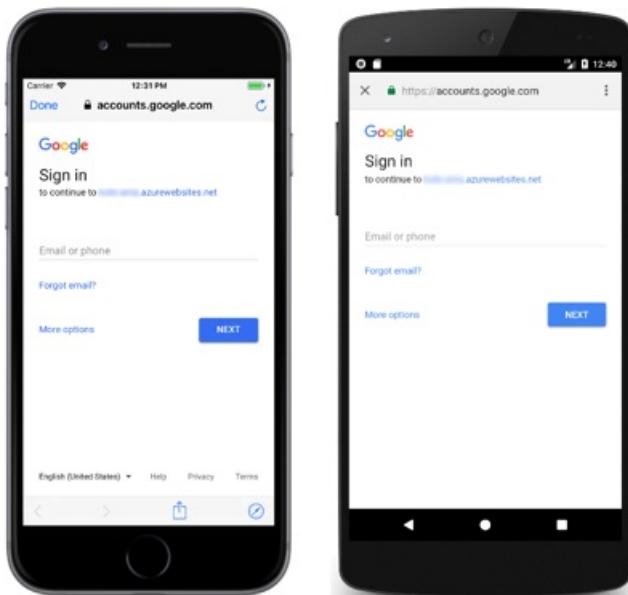
## Using an Azure Mobile Apps Instance

The [Azure Mobile Client SDK](#) provides the `MobileServiceClient` class, which is used by a Xamarin.Forms application to access the Azure Mobile Apps instance.

The sample application uses Google as the identity provider, which allows users with Google accounts to login to the Xamarin.Forms application. While Google is used as the identity provider in this article, the approach is equally applicable to other identity providers.

### Logging in Users

The login screen in the sample application is shown in the following screenshots:



While Google is used as the identity provider, a variety of other identity providers can be used, including Facebook, Microsoft, Twitter, and Azure Active Directory.

The following code example shows how the login process is invoked:

```
async void OnLoginButtonClicked(object sender, EventArgs e)
{
    ...
    if (App.Authenticator != null)
    {
        authenticated = await App.Authenticator.AuthenticateAsync();
    }
    ...
}
```

The `App.Authenticator` property is an `IAuthenticate` instance that's set by each platform-specific project. The `IAuthenticate` interface specifies an `AuthenticateAsync` operation that must be provided by each platform project. Therefore, invoking the `App.Authenticator.AuthenticateAsync` method executes the `IAuthenticate.AuthenticateAsync` method in a platform project.

All of the platform `IAuthenticate.AuthenticateAsync` methods call the `MobileServiceClient.LoginAsync` method to display a login interface and cache data. The following code example shows the `LoginAsync` method for the iOS

platform:

```
public async Task<bool> AuthenticateAsync()
{
    ...
    // The authentication provider could also be Facebook, Twitter, or Microsoft
    user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        UIApplication.SharedApplication.KeyWindow.RootViewController,
        MobileServiceAuthenticationProvider.Google,
        Constants.URLScheme);
    ...
}
```

The following code example shows the `LoginAsync` method for the Android platform:

```
public async Task<bool> AuthenticateAsync()
{
    ...
    // The authentication provider could also be Facebook, Twitter, or Microsoft
    user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        this,
        MobileServiceAuthenticationProvider.Google,
        Constants.URLScheme);
    ...
}
```

The following code example shows the `LoginAsync` method for the Universal Windows Platform:

```
public async Task<bool> AuthenticateAsync()
{
    ...
    // The authentication provider could also be Facebook, Twitter, or Microsoft
    user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        MobileServiceAuthenticationProvider.Google,
        Constants.URLScheme);
    ...
}
```

On all platforms, the `MobileServiceAuthenticationProvider` enumeration is used to specify the identity provider that will be used in the authentication process. When the `MobileServiceClient.LoginAsync` method is invoked, Azure Mobile Apps initiates an authentication flow by displaying the login page of the selected provider, and by generating an authentication token after successful login with the identity provider. The

`MobileServiceClient.LoginAsync` method returns a `MobileServiceUser` instance that will be stored in the `MobileServiceClient.CurrentUser` property. This property provides `UserId` and `MobileServiceAuthenticationToken` properties. These represent the authenticated user and an authentication token for the user. The authentication token will be included in all requests made to the Azure Mobile Apps instance, allowing the Xamarin.Forms application to perform actions on the Azure Mobile App instance that require authenticated user permissions.

## Logging Out Users

The following code example shows how the logout process is invoked:

```

async void OnLogoutButtonClicked(object sender, EventArgs e)
{
    bool loggedOut = false;

    if (App.Authenticator != null)
    {
        loggedOut = await App.Authenticator.LogoutAsync ();
    }
    ...
}

```

The `App.Authenticator` property is an `IAuthenticate` instance that's set by each platform project. The `IAuthenticate` interface specifies an `LogoutAsync` operation that must be provided by each platform project. Therefore, invoking the `App.Authenticator.LogoutAsync` method executes the `IAuthenticate.LogoutAsync` method in a platform project.

All of the platform `IAuthenticate.LogoutAsync` methods call the `MobileServiceClient.LogoutAsync` method to de-authenticate the logged-in user with the identity provider. The following code example shows the `LogoutAsync` method for the iOS platform:

```

public async Task<bool> LogoutAsync()
{
    ...
    foreach (var cookie in NSHttpCookieStorage.SharedStorage.Cookies)
    {
        NSHttpCookieStorage.SharedStorage.DeleteCookie (cookie);
    }
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
    ...
}

```

The following code example shows the `LogoutAsync` method for the Android platform:

```

public async Task<bool> LogoutAsync()
{
    ...
    CookieManager.Instance.RemoveAllCookie();
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
    ...
}

```

The following code example shows the `LogoutAsync` method for the Universal Windows Platform:

```

public async Task<bool> LogoutAsync()
{
    ...
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
    ...
}

```

When the `IAuthenticate.LogoutAsync` method is invoked, any cookies set by the identity provider are cleared, before the `MobileServiceClient.LogoutAsync` method is invoked to de-authenticate the logged-in user with the identity provider.

## Summary

This article explained how to use Azure Mobile Apps to manage the authentication process in a Xamarin.Forms

application. Azure Mobile Apps use a variety of external identity providers to support authenticating and authorizing application users, including Facebook, Google, Microsoft, Twitter, and Azure Active Directory. The `MobileServiceClient` class is used by the `Xamarin.Forms` application to control access to the Azure Mobile Apps instance.

## Related Links

- [TodoAzureAuth \(sample\)](#)
- [Consuming an Azure Mobile App](#)
- [Add authentication to your Xamarin.Forms app](#)
- [Azure Mobile Client SDK](#)
- [MobileServiceClient](#)

# Authenticating Users with Azure Active Directory B2C

6/14/2018 • 8 minutes to read • [Edit Online](#)

Azure Active Directory B2C is a cloud identity management solution for consumer-facing web and mobile applications. This article demonstrates how to use Microsoft Authentication Library and Azure Active Directory B2C to integrate consumer identity management into a mobile application.



## NOTE

The [Microsoft Authentication Library](#) is still in preview, but is suitable for use in a production environment. However, there may be breaking changes to the API, internal cache format, and other mechanisms of the library, which may impact your application.

## Overview

Azure Active Directory B2C is an identity management service for consumer-facing applications, that allows consumers to sign-in to your application by:

- Using their existing social accounts (Microsoft, Google, Facebook, Amazon, LinkedIn).
- Creating new credentials (email address and password, or username and password). These credentials are referred to as *local* accounts.

The process for integrating the Azure Active Directory B2C identity management service into a mobile application is as follows:

1. Create an Azure Active Directory B2C tenant. For more information, see [Create an Azure Active Directory B2C tenant in the Azure portal](#).
2. Register your mobile application with the Azure Active Directory B2C tenant. The registration process assigns an **Application ID** that uniquely identifies your application, and a **Redirect URL** that can be used to direct responses back to your application. For more information, see [Azure Active Directory B2C: Register your application](#).
3. Create a sign-up and sign-in policy. This policy will define the experiences that consumers will go through during sign-up and sign-in, and also specifies the contents of tokens the application will receive on successful sign-up or sign-in. For more information, see [Azure Active Directory B2C: Built-in policies](#).
4. Use the [Microsoft Authentication Library](#) (MSAL) in your mobile application to initiate an authentication workflow with your Azure Active Directory B2C tenant.

#### **NOTE**

As well as integrating Azure Active Directory B2C identity management into mobile applications, MSAL can also be used to integrate Azure Active Directory identity management into mobile applications. This can be accomplished by registering a mobile application with Azure Active Directory at the [Application Registration Portal](#). The registration process assigns an **Application ID** that uniquely identifies your application, which should be specified when using MSAL. For more information, see [How to register an app with the v2.0 endpoint](#), and [Authenticate Your Mobile Apps Using Microsoft Authentication Library](#) on the Xamarin blog.

MSAL uses the device's web browser to perform authentication. This improves the usability of an application, as users only need to sign-in once per device, improving conversion rates of sign-in and authorization flows in the application. The device browser also provides improved security. After the user completes the authentication process, control will return to the application from the web browser tab. This is achieved by registering a custom URL scheme for the redirect URL that's returned from the authentication process, and then detecting and handling the custom URL once it's sent. For more information about choosing a custom URL scheme, see [Choosing a native app redirect URI](#).

#### **NOTE**

The mechanism for registering a custom URL scheme with the operating system and handling the scheme is specific to each platform.

Each request that is sent to an Azure Active Directory B2C tenant specifies a *policy*. Policies describe consumer identity experiences such as sign-up, or sign-in. For example, a sign-up policy allows the behavior of the Azure Active Directory B2C tenant to be configured through the following settings:

- Account types that consumers can use to sign-in to the application.
- Data to be collected from the consumer during sign-up.
- Multi-factor authentication.
- Sign-up page content.
- Token claims that the mobile application receives when the policy has executed.

An Azure Active Directory tenant can contain multiple policies of different types, which can then be used in your application as required. In addition, policies can be reused across applications, allowing you to define and modify consumer identity experiences without changing your code. For more information about policies, see [Azure Active Directory B2C: Built-in policies](#).

## Setup

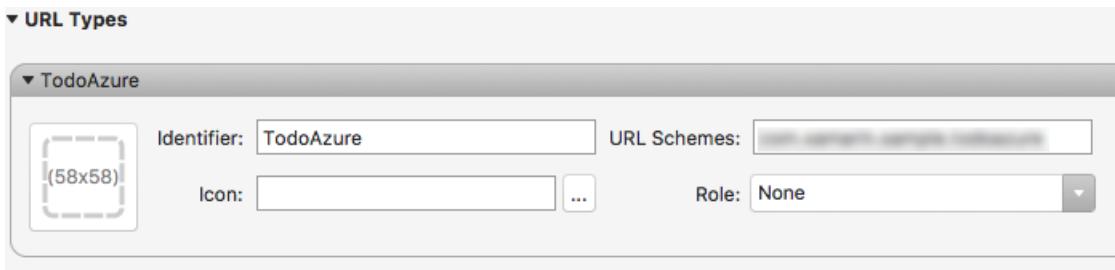
The Microsoft Authentication Library (MSAL) NuGet library must be added to the Portable Class Library (PCL) project and platform projects in a Xamarin.Forms solution. The following sections provide additional setup instructions for using MSAL to communicate with an Azure Active Directory B2C tenant from a mobile application.

### **Portable Class Library**

PCLs that consume MSAL will need to be retargeted to use Profile7. For more information about PCLs, see [Introduction to Portable Class Libraries](#).

### **iOS**

On iOS, the custom URL scheme that was registered with Azure Active Directory B2C must be registered in **Info.plist**, as shown in the following screenshot:



When Azure Active Directory B2C completes the authorization request, it redirects to the registered redirect URL. Because the URL uses a custom scheme it results in iOS launching the mobile application, passing in the URL as a launch parameter, where it's processed by the `OpenUrl` override of the application's `AppDelegate` class, which is shown in the following code example:

```
using Microsoft.Identity.Client;

namespace TodoAzure.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        ...
        public override bool OpenUrl(UIApplication app, NSUrl url, NSDictionary options)
        {
            AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url);
            return true;
        }
    }
}
```

The code in the `OpenURL` method ensures that control returns to MSAL once the interactive portion of the authentication workflow has ended.

## Android

On Android, the custom URL scheme that was registered with Azure Active Directory B2C must be registered in **AndroidManifest.xml**, by adding an `<activity>` element inside the existing `<application>` element. The `<activity>` element specifies the `IntentFilter` on the `Activity` that handles the scheme, and is shown in the following example:

```
<application ...>
    <activity android:name="microsoft.identity.client.BrowserTabActivity">
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.DEFAULT" />
            <category android:name="android.intent.category.BROWSABLE" />
            <data android:scheme="INSERT_URL_SCHEME_HERE" android:host="auth" />
        </intent-filter>
    </activity>
</application>
```

When Azure Active Directory B2C completes the authorization request, it redirects to the registered redirect URL. Because the URL uses a custom scheme it results in Android launching the mobile application, passing in the URL as a launch parameter, where it's processed by the `microsoft.identity.client.BrowserTabActivity`. Note that the `data android:scheme` property must be set to the custom URL scheme that's registered with the Azure Active Directory B2C application.

In addition, the `MainActivity` class must be modified, as shown in the following code example:

```

using Microsoft.Identity.Client;

namespace TodoAzure.Droid
{
    ...
    public class MainActivity : FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            global::Xamarin.Forms.Forms.Init(this, bundle);
            Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();
            LoadApplication(new App());
            App.UiParent = new UIParent(this);
        }

        protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
        {
            base.OnActivityResult(requestCode, resultCode, data);
            AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode, resultCode,
data);
        }
    }
}

```

The `OnCreate` method is modified by assigning a `UIParent` instance to the `App.UiParent` property. This ensures that the authentication flow occurs in the context of the current activity.

The code in the `OnActivityResult` method ensures that control returns to MSAL once the interactive portion of the authentication workflow has ended.

## Universal Windows Platform

On the Universal Windows Platform, no additional setup is required to use MSAL.

## Initialization

The Microsoft Authentication Library uses members of the `PublicClientApplication` class to initiate an authentication workflow. The sample application declares and initializes a `public` property of this type, named `ADB2CClient`, in the `AuthenticationProvider` class. The following code example shows how this property is initialized:

```

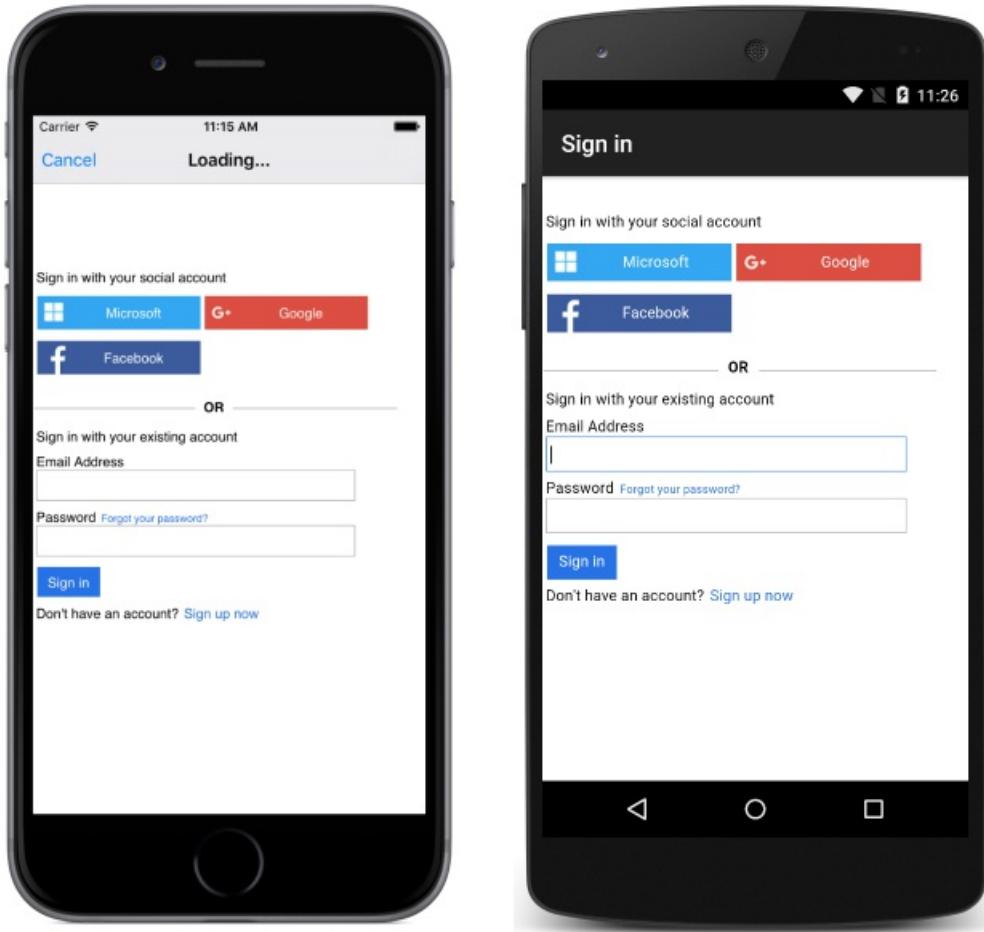
ADB2CClient = new PublicClientApplication(Constants.ClientID, Constants.Authority);

```

When the mobile application was registered with the Azure Active Directory B2C tenant, the registration process assigned an **Application ID**. This ID must be specified in the `PublicClientApplication` constructor, along with an `Authority` constant that comprises a base URL, and the Azure Active Directory B2C policy to be executed.

## Signing In

The sign-in screen in the sample application is shown in the following screenshots:



Sign-in with social identity providers, or with a local account, are permitted. While Microsoft, Google, and Facebook, as shown above, are used as social identity providers, other identity providers can also be used.

The following code example shows how the sign-in process is invoked:

```
using Microsoft.Identity.Client;

public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult = await ADB2CClient.AcquireTokenAsync(
        Constants.Scopes,
        GetUserByPolicy(ADB2CClient.Users, Constants.PolicySignUpSignIn),
        App.UiParent);
    ...
}
```

The `AcquireTokenAsync` method launches the device's web browser and displays the authentication options defined in the Azure Active Directory B2C policy that's specified by the policy referenced through the `Constants.Authority` constant. This policy defines the experiences that consumers will go through during sign-up and sign-in, and the claims the application will receive on successful sign-up or sign-in.

The result of the `AcquireTokenAsync` method call is an `AuthenticationResult` instance. If authentication is successful, the `AuthenticationResult` instance will contain an identity token, which will be cached locally. If authentication is unsuccessful, the `AuthenticationResult` instance will contain data that indicates why authentication failed.

In the sample application, if authentication is successful, the `TodoList` page is navigated to.

## Silent Re-authentication

When the `LoginPage` in the sample application appears, an attempt is made to retrieve a user token without showing any authentication user interface. This is achieved with the `AcquireTokenSilentAsync` method, as demonstrated in the following code example:

```
public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult;

    if (useSilent)
    {
        authenticationResult = await ADB2CCClient.AcquireTokenSilentAsync(
            Constants.Scopes,
            GetUserByPolicy(ADB2CCClient.Users, Constants.PolicySignUpSignIn),
            Constants.Authority,
            false);
    }
    ...
}
```

The `AcquireTokenSilentAsync` method attempts to retrieve a user token from the cache, without requiring the user to sign-in. This handles the scenario where a suitable token may already be present in the cache from previous sessions. If the attempt to obtain a token is successful, the `TodoList` page is navigated to. If the attempt to obtain a token is unsuccessful, nothing happens and the user will have the choice to initiate a new authentication workflow.

## Signing Out

The following code example shows how the sign-out process is invoked:

```
public async Task<bool> LogoutAsync()
{
    ...
    foreach (var user in ADB2CCClient.Users)
    {
        ADB2CCClient.Remove(user);
    }
    ...
}
```

This clears all the authentication tokens from the local cache.

## Summary

This article demonstrated how to use Microsoft Authentication Library (MSAL) and Azure Active Directory B2C to integrate consumer identity management into a mobile application. Azure Active Directory B2C is a cloud identity management solution for consumer-facing web and mobile applications.

## Related Links

- [AzureADB2CAuth \(sample\)](#)
- [Azure Active Directory B2C](#)
- [Microsoft Authentication Library](#)

# Integrating Azure Active Directory B2C with Azure Mobile Apps

4/12/2018 • 8 minutes to read • [Edit Online](#)

Azure Active Directory B2C is a cloud identity management solution for consumer-facing web and mobile applications. This article demonstrates how to use Azure Active Directory B2C to provide authentication and authorization to an Azure Mobile Apps instance with Xamarin.Forms.



## NOTE

The [Microsoft Authentication Library](#) is still in preview, but is suitable for use in a production environment. However, there may be breaking changes to the API, internal cache format, and other mechanisms of the library, which may impact your application.

## Overview

Azure Mobile Apps allow you to develop applications with scalable backends hosted in Azure App Service, with support for mobile authentication, offline sync, and push notifications. For more information about Azure Mobile Apps, see [Consuming an Azure Mobile App](#), and [Authenticating Users with Azure Mobile Apps](#).

Azure Active Directory B2C is an identity management service for consumer-facing applications, that allows consumers to sign-in to your application by:

- Using their existing social accounts (Microsoft, Google, Facebook, Amazon, LinkedIn).
- Creating new credentials (email address and password, or username and password). These credentials are referred to as *local* accounts.

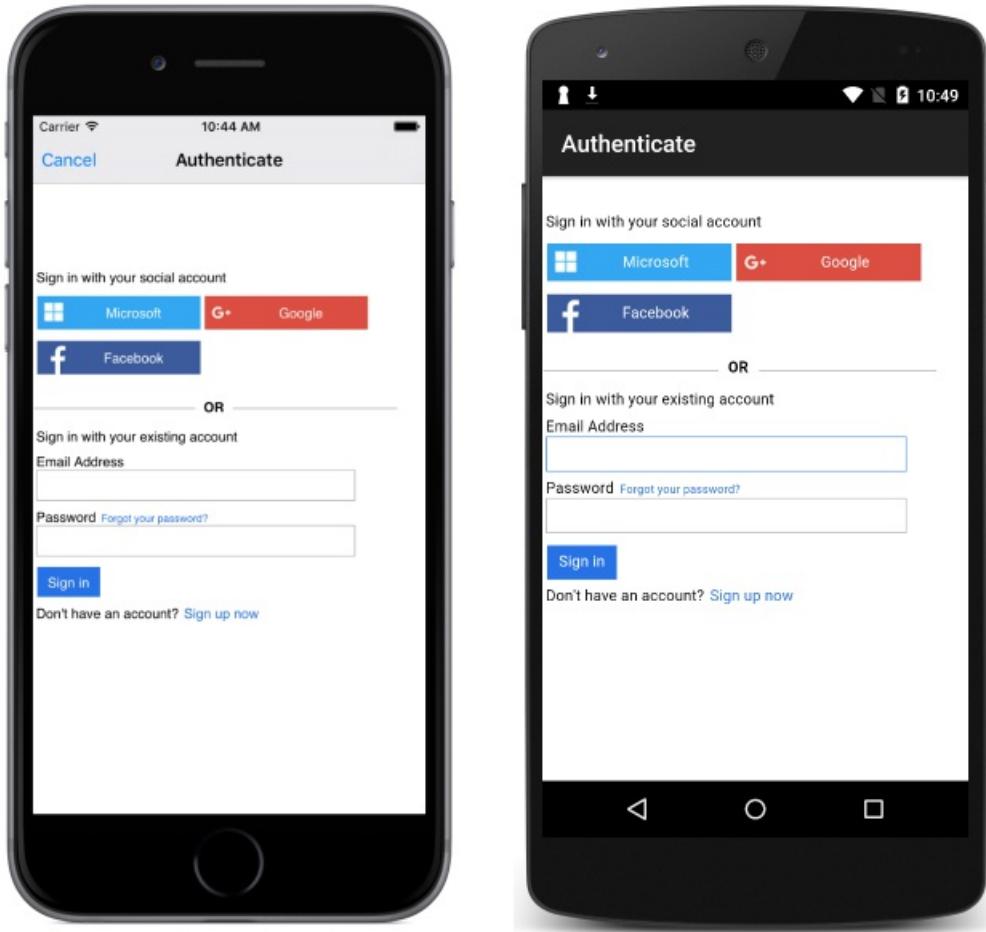
For more information about Azure Active Directory B2C, see [Authenticating Users with Azure Active Directory B2C](#).

Azure Active Directory B2C can be used to manage the authentication workflow for an Azure Mobile App. With this approach, the identity management experience is fully defined in the cloud, and can be modified without changing your mobile application code.

There are two authentication workflows that can be adopted when integrating an Azure Active Directory B2C tenant with an Azure Mobile Apps instance:

- [Client-managed](#) – in this approach the Xamarin.Forms mobile application initiates the authentication process with the Azure Active Directory B2C tenant, and passes the received authentication token to the Azure Mobile Apps instance.
- [Server-managed](#) – in this approach the Azure Mobile Apps instance uses the Azure Active Directory B2C tenant to initiate the authentication process through a web-based workflow.

In both cases, the authentication experience is provided by the Azure Active Directory B2C tenant. In the sample application, this results in the sign-in screen shown in the following screenshots:



Sign-in with social identity providers, or with a local account, are permitted. While Microsoft, Google, and Facebook are used as social identity providers in this example, other identity providers can also be used.

## Setup

Regardless of the authentication workflow used, the initial process for integrating an Azure Active Directory B2C tenant with an Azure Mobile Apps instance is as follows:

1. Create an Azure Mobile Apps instance. For more information, see [Consuming an Azure Mobile App](#).
2. Enable authentication in the Azure Mobile Apps instance and the Xamarin.Forms application. For more information, see [Authenticating Users with Azure Mobile Apps](#).
3. Create an Azure Active Directory B2C tenant. For more information, see [Authenticating Users with Azure Active Directory B2C](#).

Note that the Microsoft Authentication Library (MSAL) is required when using a client-managed authentication workflow. MSAL uses the device's web browser to perform authentication. This improves the usability of an application, as users only need to sign-in once per device, improving conversion rates of sign-in and authorization flows in the application. The device browser also provides improved security. After the user completes the authentication process, control will return to the application from the web browser tab. This is achieved by registering a custom URL scheme for the redirect URL that's returned from the authentication process, and then detecting and handling the custom URL once it's sent. For more information about using MSAL to communicate with an Azure Active Directory B2C tenant, see [Authenticating Users with Azure Active Directory B2C](#).

## Client-Managed Authentication

In client-managed authentication, a Xamarin.Forms mobile application contacts an Azure Active Directory B2C tenant to initiate an authentication flow. After successful sign-on the Azure Active Directory B2C tenant returns an identity token which is then provided during sign-in to the Azure Mobile Apps instance. This allows the

Xamarin.Forms application to perform actions on the Azure Mobile Apps instance that requires authenticated user permissions.

## Azure Active Directory B2C Tenant Configuration

For a client-managed authentication workflow, the Azure Active Directory B2C tenant should be configured as follows:

- Include a native client.
- Set the Custom Redirect URI to a URL scheme that uniquely identifies the mobile application, followed by `://auth/`. For more information about choosing a custom URL scheme, see [Choosing a native app redirect URI](#).

The following screenshot demonstrates this configuration:

\* Name

Application ID

Web App / Web API

Include web app / web API

Yes

Native client

Include native client

Yes

Redirect URI

<https://login.microsoftonline.com/tfp/oauth2/nativeclient>

Info Redirect URIs must not be http or https

Custom Redirect URI

The policy used in the Azure Active Directory B2C tenant should also be configured so that the reply URL is set to the same custom URL scheme, followed by `://auth/`. The following screenshot demonstrates this configuration:

B2C\_1\_TodoSignInAndUp X

SIGN-UP OR SIGN-IN POLICY

Edit Delete Download

[https://login.microsoftonline.com/redacted/v2.0/.well-known/openid-configuration?p=B2C\\_1\\_TodoSignInAndUp](https://login.microsoftonline.com/redacted/v2.0/.well-known/openid-configuration?p=B2C_1_TodoSignInAndUp)

---

RUN POLICY SETTINGS

Select application

Select reply url

ACCESS TOKENS

Run now endpoint

Run now

## Azure Mobile App Configuration

For a client-managed authentication workflow, the Azure Mobile Apps instance should be configured as follows:

- App Service Authentication should be turned on.
- The action to take when a request is not authenticated should be set to **Log in with Azure Active Directory**.

The following screenshot demonstrates this configuration:

Authentication / Authorization

To enable Authentication / Authorization

App Service Authentication

Off **On**

Action to take when request is not authenticated

**Log in with Azure Active Directory**

Authentication Providers

Azure Active Directory  
Configured (Advanced)

The Azure Mobile Apps instance should also be configured to communicate with the Azure Active Directory B2C tenant. This can be accomplished by enabling **Advanced** mode for the Azure Active Directory authentication provider, with the **Client ID** being the **Application ID** of the Azure Active Directory B2C tenant, and the **Issuer Url** being the metadata endpoint for the Azure Active Directory B2C policy. The following screenshot demonstrates this configuration:

Active Directory Authentication

These settings allow users to sign in with Azure Active Directory. Click here to learn more. [Learn more](#)

Management mode ⓘ **Advanced**

Client ID

Issuer Url ⓘ <https://login.microsoftonline.com/> ...

Client Secret (Optional)

**ALLOWED TOKEN AUDIENCES**

https://... .azurewebsites.net ...  
...  
...

## Signing In

The following code example shows how to initiate a client-managed authentication workflow:

```

public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult = await ADB2CClient.AcquireTokenAsync(
        Constants.Scopes,
        GetUserByPolicy(ADB2CClient.Users, Constants.PolicySignUpSignIn),
        App.UiParent);

    ...
    var payload = new JObject();
    payload["access_token"] = authenticationResult.IdToken;

    User = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory,
        payload);
    ...
}

```

The Microsoft Authentication Library (MSAL) is used to initiate an authentication workflow with the Azure Active Directory B2C tenant. The `AcquireTokenAsync` method launches the device's web browser and displays authentication options defined in the Azure Active Directory B2C policy that's specified by the policy referenced through the `Constants.Authority` constant. This policy defines the experiences that consumers will go through during sign-up and sign-in, and the claims the application will receive on successful sign-up or sign-in.

The result of the `AcquireTokenAsync` method call is an `AuthenticationResult` instance. If authentication is successful, the `AuthenticationResult` instance will contain an identity token, which will be cached locally. If authentication is unsuccessful, the `AuthenticationResult` instance will contain data that indicates why authentication failed. For information on how to use MSAL to communicate with an Azure Active Directory B2C tenant, see [Authenticating Users with Azure Active Directory B2C](#).

When the `MobileServiceClient.LoginAsync` method is invoked, the Azure Mobile Apps instance receives the identity token wrapped in a `JObject`. The presence of a valid token means that the Azure Mobile Apps instance doesn't need to initiate its own OAuth 2.0 authentication flow. Instead, the `MobileServiceClient.LoginAsync` method returns a `MobileServiceUser` instance that will be stored in the `MobileServiceClient.CurrentUser` property. This property provides `UserId` and `MobileServiceAuthenticationToken` properties. These represent the authenticated user and an authentication token for the user, which can be used until it expires. The authentication token will be included in all requests made to the Azure Mobile Apps instance, allowing the Xamarin.Forms application to perform actions on the Azure Mobile Apps instance that require authenticated user permissions.

## Signing Out

The following code example shows how the client-managed sign-out process is invoked:

```

public async Task<bool> LogoutAsync()
{
    ...
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();

    foreach (var user in ADB2CClient.Users)
    {
        ADB2CClient.Remove(user);
    }
    ...
}

```

The `MobileServiceClient.LogoutAsync` method de-authenticates the user with the Azure Mobile Apps instance, and then all authentication tokens are cleared from the local cache created by MSAL.

# Server-Managed Authentication

In server-managed authentication, a Xamarin.Forms application contacts an Azure Mobile Apps instance, which uses the Azure Active Directory B2C tenant to manage the OAuth 2.0 authentication flow by displaying a sign-in page as defined in the B2C policy. Following successful sign-on, the Azure Mobile Apps instance returns a token that allows the Xamarin.Forms application to perform actions on the Azure Mobile Apps instance that require authenticated user permissions.

## Azure Active Directory B2C Tenant Configuration

For a server-managed authentication workflow, the Azure Active Directory B2C tenant should be configured as follows:

- Include a web app/web API, and allow the implicit flow.
- Set the Reply URL to the address of the Azure Mobile App, followed by `/auth/login/aad/callback`.

The following screenshot demonstrates this configuration:

The screenshot shows the 'Web App / Web API' configuration section of the Azure Active Directory B2C tenant. It includes fields for 'Name' (TodoAzureADB2C-Server), 'Application ID' (redacted), and settings for 'Web App / Web API' (Yes selected) and 'Allow implicit flow' (Yes selected). A note indicates that 'Redirect URIs must all belong to the same domain'. The 'Reply URL' field contains `https://[redacted].onmicrosoft.com/auth/login/aad/callback`. The 'App ID URI (optional)' field contains `https://[redacted].onmicrosoft.com/`. The 'Native client' section has 'Include native client' set to No.

The policy used in the Azure Active Directory B2C tenant should also be configured so that the Reply URL is set to the address of the Azure Mobile App, followed by `/auth/login/aad/callback`. The following screenshot demonstrates this configuration:

B2C\_1\_TodoSignInAndUp-Server

SIGN-UP OR SIGN-IN POLICY

Edit Delete Download

<https://login.microsoftonline.com/> /v2.0/.well-known/openid-configuration?p=B2C\_1\_TodoSignInAndUp-Server

---

RUN POLICY SETTINGS

Select application  
TodoAzureADB2C-Server

Select reply url  
https://.azurewebsites.net/.auth/login/aad/callback

ACCESS TOKENS

Run now endpoint <https://login.microsoftonline.com/> /oauth2/v2.0/authorize?p=...

Run now

## Azure Mobile Apps Instance Configuration

For a server-managed authentication workflow, the Azure Mobile Apps instance should be configured as follows:

- App Service Authentication should be turned on.
- The action to take when a request is not authenticated should be set to **Log in with Azure Active Directory**.

The following screenshot demonstrates this configuration:

Authentication / Authorization

To enable Authentication / Authorization

App Service Authentication  
Off On

Action to take when request is not authenticated  
Log in with Azure Active Directory

Authentication Providers

Azure Active Directory  
Configured (Advanced)

The Azure Mobile Apps instance should also be configured to communicate with the Azure Active Directory B2C tenant. This can be accomplished by enabling **Advanced** mode for the Azure Active Directory authentication provider, with the **Client ID** being the **Application ID** of the Azure Active Directory B2C tenant, and the **Issuer Url** being the metadata endpoint for the Azure Active Directory B2C policy. The following screenshot demonstrates this configuration:



## Active Directory Authentication

These settings allow users to sign in with Azure Active Directory. Click here to learn more. [Learn more](#)

Management mode [?](#) [Off](#) [Express](#) [Advanced](#)

Client ID

Issuer Url [?](#)

 https://login.microsoftonline.com/ [REDACTED] /v2.0/.well-known/openid-confi...

Client Secret (Optional)

### ALLOWED TOKEN AUDIENCES

https://[REDACTED].azurewebsites.net ...

[REDACTED] ...

## Signing In

The following code example shows how to initiate a server-managed authentication workflow:

```
public async Task<bool> AuthenticateAsync()
{
    ...
    MobileServiceUser user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        UIApplication.SharedApplication.KeyWindow.RootViewController,
        MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory,
        Constants.URLScheme);
    ...
}
```

When the `MobileServiceClient.LoginAsync` method is invoked, the Azure Mobile Apps instance executes the linked Azure Active Directory B2C policy, which initiates the OAuth 2.0 authentication flow. Note that each `AuthenticateAsync` method is platform-specific. However, each `AuthenticateAsync` method uses the `MobileServiceClient.LoginAsync` method and specifies that an Azure Active Directory tenant will be used in the authentication process. For more information, see [Logging in Users](#).

The `MobileServiceClient.LoginAsync` method returns a `MobileServiceUser` instance that will be stored in the `MobileServiceClient.CurrentUser` property. This property provides `UserId` and `MobileServiceAuthenticationToken` properties. These represent the authenticated user and an authentication token for the user, which can be used until it expires. The authentication token will be included in all requests made to the Azure Mobile Apps instance, allowing the Xamarin.Forms application to perform actions on the Azure Mobile Apps instance that require authenticated user permissions.

## Signing Out

The following code example shows how the server-managed sign-out process is invoked:

```
public async Task<bool> LogoutAsync()
{
    ...
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
    ...
}
```

The `MobileServiceClient.LogoutAsync` method de-authenticates the user with the Azure Mobile Apps instance. For more information, see [Logging Out Users](#).

## Summary

This article demonstrated how to use Azure Active Directory B2C to provide authentication and authorization to an Azure Mobile Apps instance with Xamarin.Forms. Azure Active Directory B2C is a cloud identity management solution for consumer-facing web and mobile applications.

## Related Links

- [TodoAzureAuth ServerFlow \(sample\)](#)
- [TodoAzureAuth ClientFlow \(sample\)](#)
- [Consuming an Azure Mobile App](#)
- [Authenticating Users with Azure Mobile Apps](#)
- [Authenticating Users with Azure Active Directory B2C](#)
- [Microsoft Authentication Library](#)

# Synchronizing Data with Web Services

4/12/2018 • 2 minutes to read • [Edit Online](#)

*Offline sync allows users to interact with a mobile application, viewing, adding, or modifying data, even where there isn't a network connection. Changes are stored in a local database, and once the device is online, the changes can be synced with the web service.*

## Synchronizing Offline Data with Azure Mobile Apps

This article explains how to add offline sync functionality to a Xamarin.Forms application.

### Related Links

- [Introduction to Web Services](#)
- [Async Support Overview](#)

# Synchronizing Offline Data with Azure Mobile Apps

6/8/2018 • 7 minutes to read • [Edit Online](#)

*Offline sync allows users to interact with a mobile application, viewing, adding, or modifying data, even where there isn't a network connection. Changes are stored in a local database, and once the device is online, the changes can be synced with the Azure Mobile Apps instance. This article explains how to add offline sync functionality to a Xamarin.Forms application.*

## Overview

The [Azure Mobile Client SDK](#) provides the `IMobileServiceTable` interface, which represents the operations that can be performed on tables stored in the Azure Mobile Apps instance. These operations connect directly to the Azure Mobile Apps instance and will fail if the mobile device doesn't have a network connection.

To support offline sync, the Azure Mobile Client SDK supports *sync tables*, which are provided by the `IMobileServiceSyncTable` interface. This interface provides the same Create, Read, Update, Delete (CRUD) operations as the `IMobileServiceTable` interface, but the operations read from or write to a local store. The local store isn't populated with new data from the Azure Mobile Apps instance until there is a call to *pull* data. Similarly, data isn't sent to the Azure Mobile Apps instance until there is a call to *push* local changes.

Offline sync also includes support for detecting conflicts when the same record has changed in both the local store and in the Azure Mobile Apps instance, and custom conflict resolution. Conflicts can either be handled in the local store, or in the Azure Mobile Apps instance.

For more information about offline sync, see [Offline Data Sync in Azure Mobile Apps](#) and [Enable offline sync for your Xamarin.Forms mobile app](#).

## Setup

The process for integrating offline sync between a Xamarin.Forms application and an Azure Mobile Apps instance is as follows:

1. Create an Azure Mobile Apps instance. For more information, see [Consuming an Azure Mobile App](#).
2. Add the [Microsoft.Azure.Mobile.Client.SQLiteStore](#) NuGet package to all projects in the Xamarin.Forms solution.
3. (Optional) Enable authentication in the Azure Mobile Apps instance and the Xamarin.Forms application. For more information, see [Authenticating Users with Azure Mobile Apps](#).

The following section provides additional setup instructions for configuring Universal Windows Platform (UWP) projects to use the [Microsoft.Azure.Mobile.Client.SQLiteStore](#) NuGet package. No additional setup is required to use the [Microsoft.Azure.Mobile.Client.SQLiteStore](#) NuGet package on iOS and Android.

### Universal Windows Platform

To use SQLite on the Universal Windows Platform (UWP), follow these steps:

1. Install the [SQLite for the Universal Windows Platform](#) Visual Studio Extension in your development environment.
2. In the UWP project in Visual Studio, right click **References > Add Reference**, navigate to **Extensions** and add the **SQLite for Universal Windows Platform** and **Visual C++ 2015 Runtime for Universal Windows Platform Apps** extensions to the UWP project.

## Initializing the Local Store

The local store must be initialized before any sync table operations can be performed. This is achieved in the Portable Class Library (PCL) project of the Xamarin.Forms solution:

```
using Microsoft.WindowsAzure.MobileServices;
using Microsoft.WindowsAzure.MobileServices.SQLiteStore;
using Microsoft.WindowsAzure.MobileServices.Sync;

namespace TodoAzure
{
    public partial class TodoItemManager
    {
        static TodoItemManager defaultInstance = new TodoItemManager();
        IMobileServiceClient client;
        IMobileServiceSyncTable<TodoItem> todoTable;

        private TodoItemManager()
        {
            this.client = new MobileServiceClient(Constants.ApplicationURL);
            var store = new MobileServiceSQLiteStore("localstore.db");
            store.DefineTable<TodoItem>();
            this.client.SyncContext.InitializeAsync(store);
            this.todoTable = client.GetSyncTable<TodoItem>();
        }
        ...
    }
}
```

A new local SQLite database is created by the `MobileServiceSQLiteStore` class, provided that it doesn't already exist. Then, the `DefineTable<T>` method creates a table in the local store that matches the fields in the `TodoItem` type, provided that it doesn't already exist.

A *sync context* is associated with a `MobileServiceClient` instance, and tracks changes that are made with sync tables. The sync context maintains a queue that keeps an ordered list of Create, Update, and Delete (CUD) operations that will be sent to the Azure Mobile Apps instance later. The `IMobileServiceSyncContext.InitializeAsync()` method is used to associate the local store with the sync context.

The `todoTable` field is an `IMobileServiceSyncTable`, and so all CRUD operations use the local store.

## Performing Synchronization

The local store is synchronized with the Azure Mobile Apps instance when the `SyncAsync` method is invoked:

```

public async Task SyncAsync()
{
    ReadOnlyCollection<MobileServiceTableOperationError> syncErrors = null;

    try
    {
        await this.client.SyncContext.PushAsync();

        // The first parameter is a query name that is used internally by the client SDK to implement incremental
        sync.
        // Use a different query name for each unique query in your program.
        await this.todoTable.PullAsync("allTodoItems", this.todoTable.CreateQuery());
    }
    catch (MobileServicePushFailedException exc)
    {
        if (exc.PushResult != null)
        {
            syncErrors = exc.PushResult.Errors;
        }
    }

    // Simple error/conflict handling.
    if (syncErrors != null)
    {
        foreach (var error in syncErrors)
        {
            if (error.OperationKind == MobileServiceTableOperationKind.Update && error.Result != null)
            {
                // Update failed, revert to server's copy
                await error.CancelAndUpdateItemAsync(error.Result);
            }
            else
            {
                // Discard local change
                await error.CancelAndDiscardItemAsync();
            }

            Debug.WriteLine(@"Error executing sync operation. Item: {0} ({1}). Operation discarded.",
error.TableName, error.Item["id"]);
        }
    }
}

```

The `IMobileServiceSyncTable.PushAsync` method operates on the sync context, rather than a specific table, and sends all CUD changes since the last push.

Pull is performed by the `IMobileServiceSyncTable.PullAsync` method on a single table. The first parameter to the `PullAsync` method is a query name that is used only on the mobile device. Providing a non-null query name results in the Azure Mobile Client SDK performing an *incremental sync*, where each time a pull operation returns results, the latest `updatedAt` timestamp from the results is stored in the local system tables. Subsequent pull operations then only retrieve records after that timestamp. Alternatively, *full sync* can be achieved by passing `null` as the query name, which results in all records being retrieved on each pull operation. Following any sync operation, received data is inserted into the local store.

#### **NOTE**

If a pull is executed against a table that has pending local updates, the pull will first execute a push on the sync context. This minimizes conflicts between changes that are already queued and new data from the Azure Mobile Apps instance.

The `SyncAsync` method also includes a basic implementation for handling conflicts when the same record has changed in both the local store and in the Azure Mobile Apps instance. When the conflict is that data has been

updated both in the local store and in the Azure Mobile Apps instance, the `SyncAsync` method updates the data in the local store from the data stored in the Azure Mobile Apps instance. When any other conflict occurs, the `SyncAsync` method discards the local change. This handles the scenario where a local change exists for data that's been deleted from the Azure Mobile Apps instance.

In a production application, developers should write a custom `IMobileServiceSyncHandler` conflict-handling implementation that's suited to their use case. For more information, see [Use Optimistic Concurrency for conflict resolution](#) on the Azure portal, and [Deep dive on the offline support in the managed client SDK](#) on MSDN blogs.

## Purging Data

Tables in the local store can be cleared of data with the `IMobileServiceSyncTable.PurgeAsync` method. This method supports scenarios such as removing stale data that an application no longer requires. For example, the sample application only displays `TodoItem` instances that aren't complete. Therefore, completed items no longer need to be stored locally. Purging completed items from the local store can be accomplished as follows:

```
await todoTable.PurgeAsync(todoTable.Where(item => item.Done));
```

A call to `PurgeAsync` also triggers a push operation. Therefore, any items that are marked as complete locally will be sent to the Azure Mobile Apps instance before being removed from the local store. However, if there are operations pending synchronization with the Azure Mobile Apps instance, the purge will throw an `InvalidOperationException` unless the `force` parameter is set to `true`. An alternative strategy is to examine the `IMobileServiceSyncContext.PendingOperations` property, which returns the number of pending operations that haven't been pushed to the Azure Mobile Apps instance, and only perform the purge if the property is zero.

### NOTE

Invoking `PurgeAsync` with the `force` parameter set to `true` will lose any pending changes.

## Initiating Synchronization

In the sample application, the `SyncAsync` method is invoked through the `TodoList.OnAppearing` method:

```
protected override async void OnAppearing()
{
    base.OnAppearing();

    // Set syncItems to true to synchronize the data on startup when running in offline mode
    await RefreshItems(true, syncItems: true);
}
```

This means that the application will attempt to sync with the Azure Mobile Apps instance when it starts.

In addition, sync can be initiated in iOS and Android by using pull to refresh on the list of data, and on the Windows platforms by using the **Sync** button on the user interface. For more information, see [Pull to Refresh](#).

## Summary

This article explained how to add offline sync functionality to a Xamarin.Forms application. Offline sync allows users to interact with a mobile application, viewing, adding, or modifying data, even where there isn't a network connection. Changes are stored in a local database, and once the device is online, the changes can be synced with the Azure Mobile Apps instance.

## Related Links

- [TodoAzureAuthOfflineSync \(sample\)](#)
- [Consuming an Azure Mobile App](#)
- [Authenticating Users with Azure Mobile Apps](#)
- [Azure Mobile Client SDK](#)
- [MobileServiceClient](#)

# Sending Push Notifications

4/12/2018 • 2 minutes to read • [Edit Online](#)

*A push notification is used to deliver information, such as a message, from a backend system to an application on a mobile device to increase application engagement and usage. The notification can be sent at anytime, even when the user is not actively using the targeted application.*

## Sending Push Notifications from Azure Mobile Apps

Azure Notification Hubs provide a scalable push infrastructure for sending mobile push notifications from any backend to any mobile platform, while eliminating the complexity of a backend having to communicate with different platform notification systems.

# Sending Push Notifications from Azure Mobile Apps

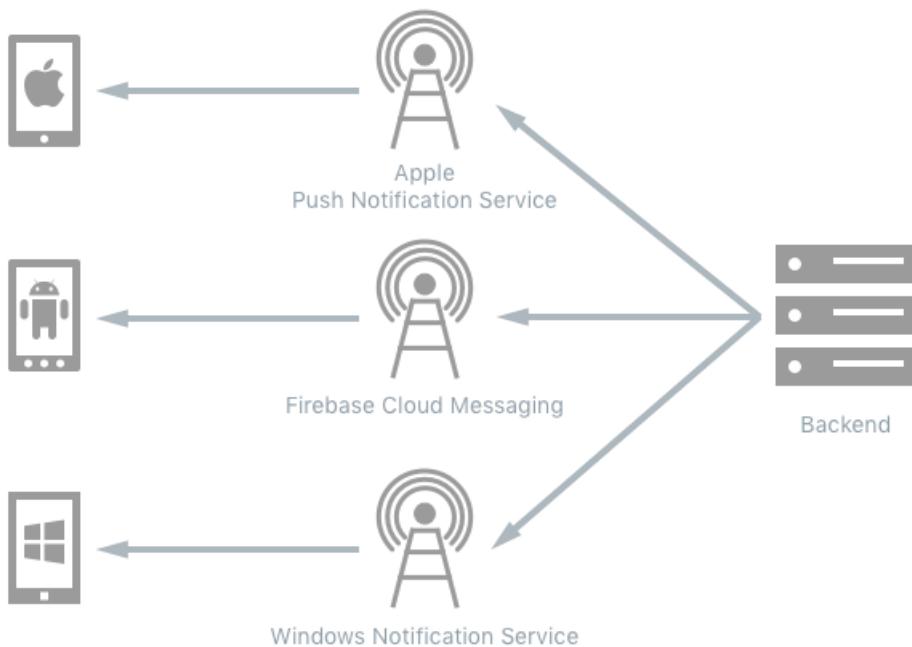
6/8/2018 • 12 minutes to read • [Edit Online](#)

Azure Notification Hubs provide a scalable push infrastructure for sending mobile push notifications from any backend to any mobile platform, while eliminating the complexity of a backend having to communicate with different platform notification systems. This article explains how to use Azure Notification Hubs to send push notifications from an Azure Mobile Apps instance to a Xamarin.Forms application.

## Azure Push Notification Hub and Xamarin.Forms, by [Xamarin University](#)

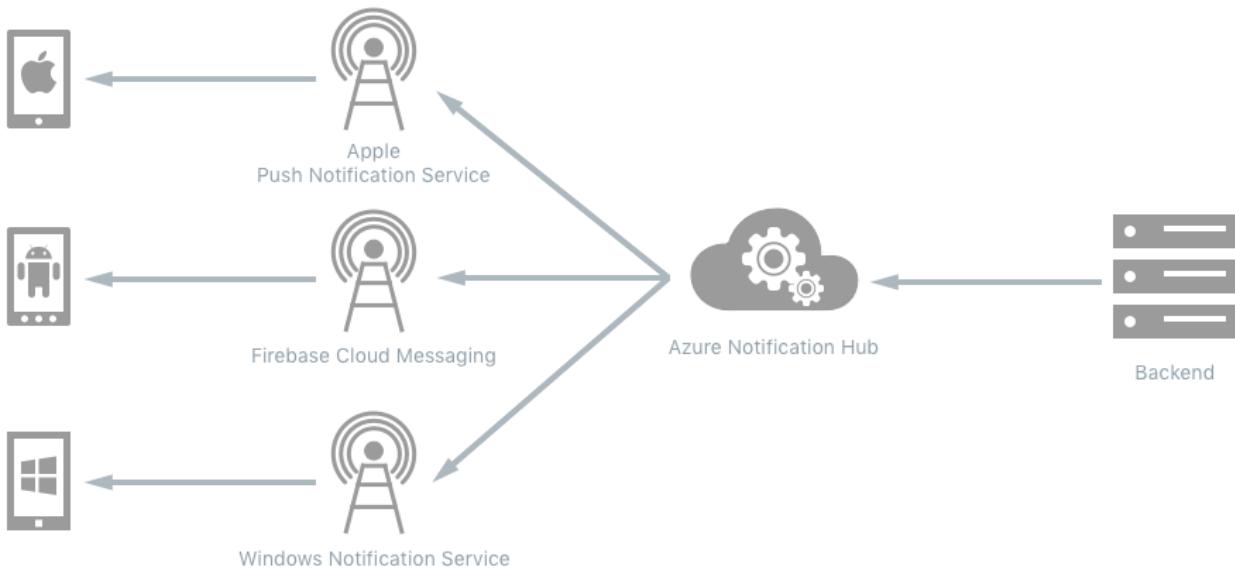
A push notification is used to deliver information, such as a message, from a backend system to an application on a mobile device to increase application engagement and usage. The notification can be sent at anytime, even when the user is not actively using the targeted application.

Backend systems send push notifications to mobile devices through Platform Notification Systems (PNS), as shown in the following diagram:



To send a push notification, the backend system contacts the platform-specific PNS to send a notification to a client application instance. This significantly increases the complexity of the backend when cross-platform push notifications are required, because the backend must use each platform-specific PNS API and protocol.

Azure Notification Hubs eliminate this complexity by abstracting the details of the different platform notification systems, allowing a cross-platform notification to be sent with a single API call, as shown in the following diagram:

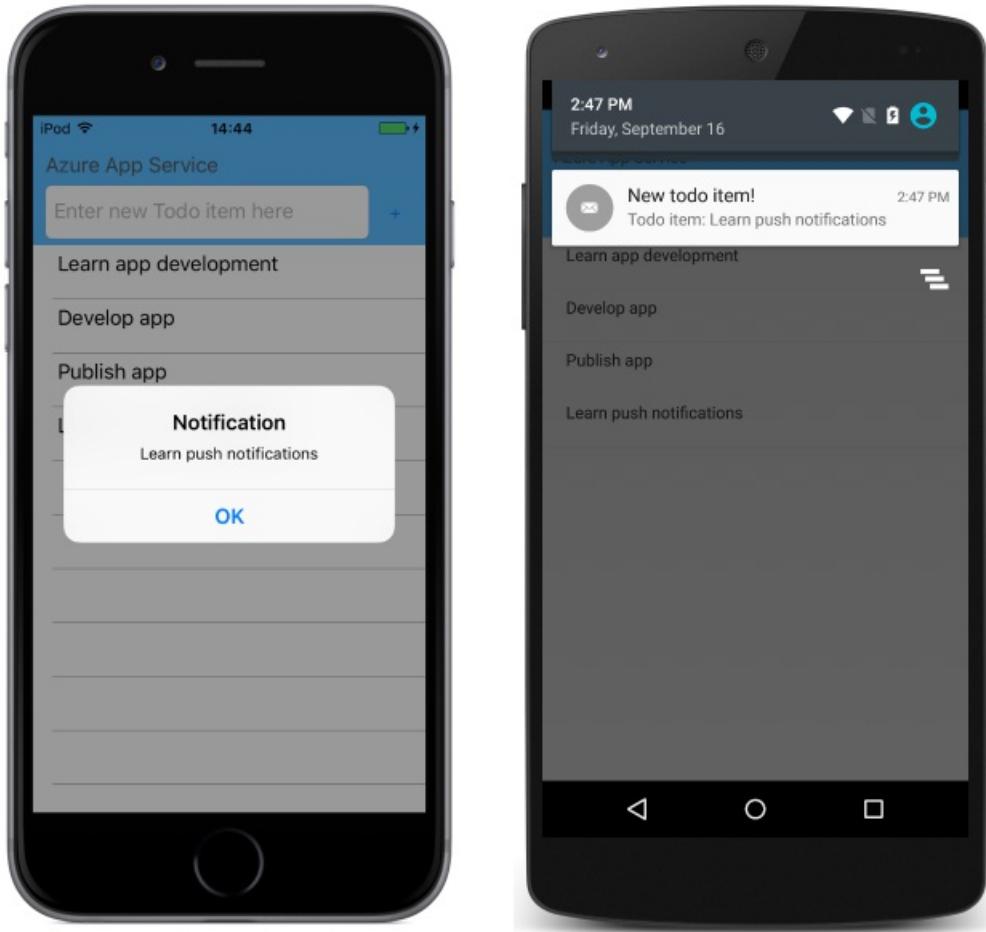


To send a push notification, the backend system only contacts the Azure Notification Hub, which in turn communicates with the different platform notification systems, therefore decreasing the complexity of the backend code that sends push notifications.

Azure Mobile Apps have built-in support for push notifications using notification hubs. The process for sending a push notification from an Azure Mobile Apps instance to a Xamarin.Forms application is as follows:

1. The Xamarin.Forms application registers with the PNS, which returns a handle.
2. The Azure Mobile Apps instance sends a notification to its Azure Notification Hub, specifying the handle of the device to be targeted.
3. The Azure Notification Hub sends the notification to the appropriate PNS for the device.
4. The PNS sends the notification to the specified device.
5. The Xamarin.Forms application processes the notification and displays it.

The sample application demonstrates a todo list application whose data is stored in an Azure Mobile Apps instance. Every time a new item is added to the Azure Mobile Apps instance, a push notification is sent to the Xamarin.Forms application. The following screenshots show each platform displaying the received push notification:



For more information about Azure Notification Hubs, see [Azure Notification Hubs](#) and [Add push notifications to your Xamarin.Forms app](#).

## Azure and Platform Notification System Setup

The process for integrating an Azure Notification Hub into an Azure Mobile Apps instance is as follows:

1. Create an Azure Mobile Apps instance. For more information, see [Consuming an Azure Mobile App](#).
2. Configure a notification hub. For more information, see [Configure a notification hub](#).
3. Update the Azure Mobile Apps instance to send push notifications. For more information, see [Update the server project to send push notifications](#).
4. Register with each PNS.
5. Configure the notification hub to communicate with each PNS.

The following sections provide additional setup instructions for each platform.

### iOS

The following additional steps must be carried out to use Apple Push Notification Service (APNS) from an Azure Notification Hub:

1. Generate a certificate signing request for the push certificate with the Keychain Access tool. For more information, see [Generate the Certificate Signing Request file for the push certificate](#) on the Azure Documentation Center.
2. Register the Xamarin.Forms application for push notification support on the Apple Developer Center. For more information, see [Register your app for push notifications](#) on the Azure Documentation Center.
3. Create a push notifications enabled provisioning profile for the Xamarin.Forms application on the Apple Developer Center. For more information, see [Create a provisioning profile for the app](#) on the Azure Documentation Center.

4. Configure the notification hub to communicate with APNS. For more information, see [Configure the notification hub for APNS](#).
5. Configure the Xamarin.Forms application to use the new App ID and provisioning profile. For more information, see [Configuring the iOS project in Xamarin Studio](#) or [Configuring the iOS project in Visual Studio](#) on the Azure Documentation Center.

## Android

The following additional steps must be carried out to use Firebase Cloud Messaging (FCM) from an Azure Notification Hub:

1. Register for FCM. A Server API key and a Client ID are automatically generated, and packed in a `google-services.json` file that is downloaded. For more information, see [Enable Firebase Cloud Messaging \(FCM\)](#).
2. Configure the notification hub to communicate with FCM. For more information, see [Configure the Mobile Apps back end to send push requests by using FCM](#).

## Universal Windows Platform

The following additional steps must be carried out to use the Windows Notification Service (WNS) from an Azure Notification Hub:

1. Register for the Windows Notification Service (WNS). For more information, see [Register your Windows app for push notifications with WNS](#) on the Azure Documentation Center.
2. Configure the notification hub to communicate with WNS. For more information, see [Configure the notification hub for WNS](#) on the Azure Documentation Center.

# Adding Push Notification Support to the Xamarin.Forms Application

The following sections discuss the implementation required in each platform-specific project to support push notifications.

## iOS

The process for implementing push notification support in an iOS application is as follows:

1. Register with the Apple Push Notification Service (APNS) in the `AppDelegate.FinishedLaunching` method. For more information, see [Registering with the Apple Push Notification System](#).
2. Implement the `AppDelegate.RegisteredForRemoteNotifications` method to handle the registration response. For more information, see [Handling the Registration Response](#).
3. Implement the `AppDelegate.DidReceiveRemoteNotification` method to process incoming push notifications. For more information, see [Processing Incoming Push Notifications](#).

## Registering with the Apple Push Notification Service

Before an iOS application can receive push notifications, it must register with the Apple Push Notification Service (APNS), which will generate a unique device token and return it to the application. Registration is invoked in the `FinishedLaunching` override in the `AppDelegate` class:

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    ...
    var settings = UIUserNotificationSettings.GetSettingsForTypes(
        UIUserNotificationType.Alert, new NSSet());
    UIApplication.SharedApplication.RegisterUserNotificationSettings(settings);
    UIApplication.SharedApplication.RegisterForRemoteNotifications();
    ...
}
```

When an iOS application registers with APNS it must specify the types of push notifications it would like to receive. The `RegisterUserNotificationSettings` method registers the types of notifications the application can receive, with the `RegisterForRemoteNotifications` method registering to receive push notifications from APNS.

#### NOTE

Failing to call the `RegisterUserNotificationSettings` method will result in push notifications being silently received by the application.

### Handling the Registration Response

The APNS registration request occurs in the background. When the response is received, iOS will call the `RegisteredForRemoteNotifications` override in the `AppDelegate` class:

```
public override void RegisteredForRemoteNotifications(UIApplication application, NSData deviceToken)
{
    const string templateBodyAPNS = "{\"aps\":{\"alert\":\"$(messageParam)\"}}";

    JObject templates = new JObject();
    templates["genericMessage"] = new JObject
    {
        {"body", templateBodyAPNS}
    };

    // Register for push with the Azure mobile app
    Push push = TodoItemManager.DefaultManager.CurrentClient.GetPush();
    push.RegisterAsync(deviceToken, templates);
}
```

This method creates a simple notification message template as JSON, and registers the device to receive template notifications from the notification hub.

#### NOTE

The `FailedToRegisterForRemoteNotifications` override should be implemented to handle situations such as no network connection. This is important because users might start the application while offline.

### Processing Incoming Push Notifications

The `DidReceiveRemoteNotification` override in the `AppDelegate` class is used to process incoming push notifications when the application is running, and is invoked when a notification is received:

```

public override void DidReceiveRemoteNotification(
    UIApplication application, NSDictionary userInfo, Action<UIBackgroundFetchResult> completionHandler)
{
    NSDictionary aps = userInfo.ObjectForKey(new NSString("aps")) as NSDictionary;

    string alert = string.Empty;
    if (aps.ContainsKey(new NSString("alert")))
        alert = (aps[new NSString("alert")] as NSString).ToString();

    // Show alert
    if (!string.IsNullOrEmpty(alert))
    {
        var notificationAlert = UIAlertController.Create("Notification", alert, UIAlertControllerStyle.Alert);
        notificationAlert.AddAction(UIAlertAction.Create("OK", UIAlertActionStyle.Cancel, null));
        UIApplication.SharedApplication.KeyWindow.RootViewController.PresentViewController(notificationAlert,
true, null);
    }
}

```

The `userInfo` dictionary contains the `aps` key, whose value is the `alert` dictionary with the remaining notification data. This dictionary is retrieved, with the `string` notification message being displayed in a dialog box.

#### **NOTE**

If an application isn't running when a push notification arrives, the application will be launched but the `DidReceiveRemoteNotification` method won't process the notification. Instead, get the notification payload and respond appropriately from the `WillFinishLaunching` or `FinishedLaunching` overrides.

For more information about APNS, see [Push Notifications in iOS](#).

## **Android**

The process for implementing push notification support in an Android application is as follows:

1. Add the [Xamarin.Firebase.Messaging](#) NuGet package to the Android project, and set the application's target version to Android 7.0 or higher.
2. Add the `google-services.json` file, downloaded from the Firebase console, to the root of the Android project and set its build action to **GoogleServicesJson**. For more information, see [Add the Google Services JSON File](#).
3. Register with Firebase Cloud Messaging (FCM) by declaring a receiver in the Android manifest file, and by implementing the `FirebaseRegistrationService.OnTokenRefresh` method. For more information, see [Registering with Firebase Cloud Messaging](#).
4. Register with the Azure Notification Hub in the `AzureNotificationHubService.RegisterAsync` method. For more information, see [Registering with the Azure Notification Hub](#).
5. Implement the `FirebaseNotificationService.OnMessageReceived` method to process incoming push notifications. For more information, see [Displaying the Contents of a Push Notification](#).

For more information about Firebase Cloud Messaging, see [Firebase Cloud Messaging](#) and [Remote Notifications with Firebase Cloud Messaging](#).

#### **Registering with Firebase Cloud Messaging**

Before an Android application can receive push notifications, it must register with FCM, which will generate a registration token and return it to the application. For more information about registration tokens, see [Registration with FCM](#).

This is accomplished by:

- Declaring a receiver in the Android manifest. For more information, see [Declaring the Receiver in the Android](#)

## Manifest

- Implementing the Firebase Instance ID Service. For more information, see [Implementing the Firebase Instance ID Service](#).

Declaring the Receiver in the Android Manifest

Edit `AndroidManifest.xml` and insert the following `<receiver>` elements into the `<application>` element:

```
<receiver android:name="com.google.firebaseio.iid.FirebaseInstanceIdInternalReceiver" android:exported="false">
</receiver>
<receiver android:name="com.google.firebaseio.iid.FirebaseInstanceIdReceiver" android:exported="true"
    android:permission="com.google.android.c2dm.permission.SEND">
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
        <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
        <category android:name="${applicationId}" />
    </intent-filter>
</receiver>
```

This XML performs the following operations:

- Declares an internal `FirebaseInstanceIdInternalReceiver` implementation that is used to start services securely.
- Declares a `FirebaseInstanceIdReceiver` implementation that provides a unique identifier for each app instance. This receiver also authenticates and authorizes actions.

The `FirebaseInstanceIdReceiver` receives `FirebaseInstanceId` and `FirebaseMessaging` events and delivers them to the class that's derived from `FirebaseInstanceIdService`.

Implementing the Firebase Instance ID Service

Registering the application with FCM is achieved by deriving a class from the `FirebaseInstanceIdService` class.

This class is responsible for generating security tokens that authorize the client application to access FCM. In the sample application the `FirebaseRegistrationService` class derives from the `FirebaseInstanceIdService` class and is shown in the following code example:

```
[Service]
[IntentFilter(new[] { "com.google.firebaseio.INSTANCE_ID_EVENT" })]
public class FirebaseRegistrationService : FirebaseInstanceIdService
{
    const string TAG = "FirebaseRegistrationService";

    public override void OnTokenRefresh()
    {
        var refreshedToken = FirebaseInstanceId.Instance.Token;
        Log.Debug(TAG, "Refreshed token: " + refreshedToken);
        SendRegistrationTokenToAzureNotificationHub(refreshedToken);
    }

    void SendRegistrationTokenToAzureNotificationHub(string token)
    {
        // Update notification hub registration
        Task.Run(async () =>
        {
            await
        AzureNotificationHubService.RegisterAsync(TodoItemManager.DefaultManager.CurrentClient.GetPush(), token);
        });
    }
}
```

The `OnTokenRefresh` method is invoked when the application receives a registration token from FCM. The method retrieves the token from the `FirebaseInstanceId.Instance.Token` property, which is asynchronously updated by FCM. The `OnTokenRefresh` method is infrequently invoked, because the token is only updated when the application

is installed or uninstalled, when the user deletes application data, when the application erases the Instance ID, or when the security of the token has been compromised. In addition, the FCM Instance ID service will request that the application refreshes its token periodically, typically every 6 months.

The `OnTokenRefresh` method also invokes the `SendRegistrationTokenToAzureNotificationHub` method, which is used to associate the user's registration token with the Azure Notification Hub.

#### Registering with the Azure Notification Hub

The `AzureNotificationHubService` class provides the `RegisterAsync` method, which associates the user's registration token with the Azure Notification Hub. The following code example shows the `RegisterAsync` method, which is invoked by the `FirebaseRegistrationService` class when the user's registration token changes:

```
public class AzureNotificationHubService
{
    const string TAG = "AzureNotificationHubService";

    public static async Task RegisterAsync(Push push, string token)
    {
        try
        {
            const string templateBody = "{\"data\":{\"message\":\"$(messageParam)\"}}";
            JObject templates = new JObject();
            templates["genericMessage"] = new JObject
            {
                {"body", templateBody}
            };

            await push.RegisterAsync(token, templates);
            Log.Info("Push Installation Id: ", push.InstallationId.ToString());
        }
        catch (Exception ex)
        {
            Log.Error(TAG, "Could not register with Notification Hub: " + ex.Message);
        }
    }
}
```

This method creates a simple notification message template as JSON, and registers to receive template notifications from the notification hub, using the Firebase registration token. This ensures that any notifications sent from the Azure Notification Hub will target the device represented by the registration token.

#### Displaying the Contents of a Push Notification

Displaying the contents of a push notification is achieved by deriving a class from the `FirebaseMessagingService` class. This class includes the overridable `OnMessageReceived` method, which is invoked when the application receives a notification from FCM, provided that the application is running in the foreground. In the sample application the `FirebaseNotificationService` class derives from the `FirebaseMessagingService` class, and is shown in the following code example:

```

[Service]
[IntentFilter(new[] { "com.google.firebaseio.MESSAGING_EVENT" })]
public class FirebaseNotificationService : FirebaseMessagingService
{
    const string TAG = "FirebaseNotificationService";

    public override void OnMessageReceived(RemoteMessage message)
    {
        Log.Debug(TAG, "From: " + message.From);

        // Pull message body out of the template
        var messageBody = message.Data["message"];
        if (string.IsNullOrWhiteSpace(messageBody))
            return;

        Log.Debug(TAG, "Notification message body: " + messageBody);
        SendNotification(messageBody);
    }

    void SendNotification(string messageBody)
    {
        var intent = new Intent(this, typeof(MainActivity));
        intent.AddFlags(ActivityFlags.ClearTop);
        var pendingIntent = PendingIntent.GetActivity(this, 0, intent, PendingIntentFlags.OneShot);

        var notificationBuilder = new NotificationCompat.Builder(this)
            .SetSmallIcon(Resource.Drawable.ic_stat_ic_notification)
            .SetContentTitle("New Todo Item")
            .SetContentText(messageBody)
            .SetContentIntent(pendingIntent)
            .SetSound(RingtoneManager.GetDefaultUri(RingtoneType.Notification))
            .SetAutoCancel(true);

        var notificationManager = NotificationManager.FromContext(this);
        notificationManager.Notify(0, notificationBuilder.Build());
    }
}

```

When the application receives a notification from FCM, the `OnMessageReceived` method extracts the message content, and calls the `SendNotification` method. This method converts the message content into a local notification that's launched while the application is running, with the notification appearing in the notification area.

#### Handling Notification Intents

When a user taps a notification, any data accompanying the notification message is made available in the `Intent` extras. This data can be extracted with the following code:

```

if (Intent.Extras != null)
{
    foreach (var key in Intent.Extras.KeySet())
    {
        var value = Intent.Extras.GetString(key);
        Log.Debug(TAG, "Key: {0} Value: {1}", key, value);
    }
}

```

The application's launcher `Intent` is fired when the user taps its notification message, so this code will log any accompanying data in the `Intent` to the output window.

## Universal Windows Platform

Before a Universal Windows Platform (UWP) application can receive push notifications it must register with the Windows Notification Service (WNS), which will return a notification channel. Registration is invoked by the `InitNotificationsAsync` method in the `App` class:

```

private async Task InitNotificationsAsync()
{
    var channel = await PushNotificationChannelManager
        .CreatePushNotificationChannelForApplicationAsync();

    const string templateBodyWNS =
        "<toast><visual><binding template=\"ToastText01\"><text id=\"1\">$({messageParam})</text></binding>
</visual></toast>";

    JObject headers = new JObject();
    headers["X-WNS-Type"] = "wns/toast";

    JObject templates = new JObject();
    templates["genericMessage"] = new JObject
    {
        {"body", templateBodyWNS},
        {"headers", headers} // Needed for WNS.
    };

    await TodoItemManager.DefaultManager.CurrentClient.GetPush()
        .RegisterAsync(channel.Uri, templates);
}

```

This method gets the push notification channel, creates a notification message template as JSON, and registers the device to receive template notifications from the notification hub.

The `InitNotificationsAsync` method is invoked from the `OnLaunched` override in the `App` class:

```

protected override async void OnLaunched(LaunchActivatedEventArgs e)
{
    ...
    await InitNotificationsAsync();
}

```

This ensures that the push notification registration is created or refreshed every time the application is launched, therefore ensuring that the WNS push channel is always active.

When a push notification is received it will automatically be displayed as a *toast* – a modeless window containing the message.

## Summary

This article demonstrated how to use Azure Notification Hubs to send push notifications from an Azure Mobile Apps instance to a Xamarin.Forms application. Azure Notification Hubs provide a scalable push infrastructure for sending mobile push notifications from any backend to any mobile platform, while eliminating the complexity of a backend having to communicate with different platform notification systems.

## Related Links

- [Consuming an Azure Mobile App](#)
- [Azure Notification Hubs](#)
- [Add push notifications to your Xamarin.Forms app](#)
- [Push Notifications in iOS](#)
- [Firebase Cloud Messaging](#)
- [TodoAzurePush \(sample\)](#)
- [Azure Mobile Client SDK](#)

# Storing Files in the Cloud

6/8/2018 • 2 minutes to read • [Edit Online](#)

*Azure Storage is a scalable cloud storage solution that can be used for storing unstructured, and structured data.*

## Storing Files in Azure Storage

This article demonstrates how to use Xamarin.Forms to store text and binary data in Azure Storage, and how to access the data.

# Storing and Accessing Data in Azure Storage

4/12/2018 • 9 minutes to read • [Edit Online](#)

Azure Storage is a scalable cloud storage solution that can be used to store unstructured, and structured data. This article demonstrates how to use Xamarin.Forms to store text and binary data in Azure Storage, and how to access the data.

## Overview

Azure Storage provides four storage services:

- Blob Storage. A blob can be text or binary data, such as backups, virtual machines, media files, or documents.
- Table Storage is a NoSQL key-attribute store.
- Queue Storage is a messaging service for workflow processing and communication between cloud services.
- File Storage provides shared storage using the SMB protocol.

There are two types of storage accounts:

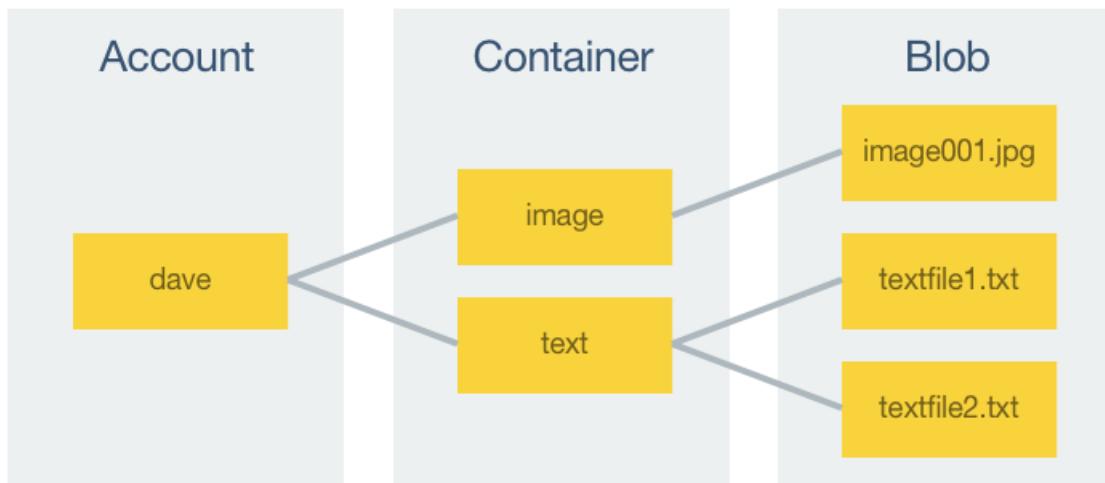
- A general-purpose storage accounts provides access to Azure Storage services from a single account.
- A Blob storage account is a specialized storage account for storing blobs. This account type is recommended when you only need to store blob data.

This article, and accompanying sample application, demonstrates uploading image and text files to blob storage, and downloading them. In addition, it also demonstrates retrieving a list of files from blob storage, and deleting files.

For more information about Azure Storage, see [Introduction to Storage](#).

## Introduction to Blob Storage

Blob storage consists of three components, which are shown in the following diagram:



All access to Azure Storage is through a storage account. A storage account can contain an unlimited number of containers, and a container can store an unlimited number of blobs, up to the capacity limit of the storage account.

A blob is a file of any type and size. Azure Storage supports three different blob types:

- Block blobs are optimized for streaming and storing cloud objects, and are a good choice for storing backups, media files, documents etc. Block blobs can be up to 195Gb in size.
- Append blobs are similar to block blobs but are optimized for append operations, such as logging. Append blobs can be up to 195Gb in size.
- Page blobs are optimized for frequent read/write operations and are typically used for storing virtual machines, and their disks. Page blobs can be up to 1Tb in size.

#### **NOTE**

Note that blob storage accounts support block and append blobs, but not page blobs.

A blob is uploaded to Azure Storage, and downloaded from Azure Storage, as a stream of bytes. Therefore, files must be converted to a stream of bytes prior to upload, and converted back to their original representation after download.

Every object that's stored in Azure Storage has a unique URL address. The storage account name forms the subdomain of that address, and the combination of subdomain and domain name forms an *endpoint* for the storage account. For example, if your storage account is named *mystorageaccount*, the default blob endpoint for the storage account is `https://mystorageaccount.blob.core.windows.net`.

The URL for accessing an object in a storage account is built by appending the object's location in the storage account to the endpoint. For example, a blob address will have the format

`https://mystorageaccount.blob.core.windows.net/mycontainer/myblob`.

## Setup

The process for integrating an Azure Storage account into a Xamarin.Forms application is as follows:

1. Create a storage account. For more information, see [Create a storage account](#).
2. Add the [Azure Storage Client Library](#) to the Xamarin.Forms application.
3. Configure the storage connection string. For more information, see [Connecting to Azure Storage](#).
4. Add `using` directives for the `Microsoft.WindowsAzure.Storage` and `Microsoft.WindowsAzure.Storage.Blob` namespaces to classes that will access Azure Storage.

#### **NOTE**

While this sample uses a Shared Access Project, the Azure Storage Client Library now also supports being consumed from a Portable Class Library (PCL) project.

## Connecting to Azure Storage

Every request made against storage account resources must be authenticated. While blobs can be configured to support anonymous authentication, there are two main approaches an application can use to authenticate with a storage account:

- Shared Key. This approach uses the Azure Storage account name and account key to access storage services. A storage account is assigned two private keys on creation that can be used for shared key authentication.
- Shared Access Signature. This is a token that can be appended to a URL that enables delegated access to a storage resource, with the permissions it specifies, for the period of time that it's valid.

Connection strings can be specified that include the authentication information required to access Azure Storage resources from an application. In addition, a connection string can be configured to connect to the Azure Storage emulator from Visual Studio.

**NOTE**

Azure Storage supports HTTP and HTTPS in a connection string. However, using HTTPS is recommended.

## Connecting to the Azure Storage Emulator

The Azure Storage emulator provides a local environment that emulates the Azure blob, queue, and table services for development purposes.

The following connection string should be used to connect to the Azure Storage emulator:

```
UseDevelopmentStorage=true
```

For more information about the Azure Storage Emulator, see [Use the Azure Storage Emulator for Development and testing](#).

## Connecting to Azure Storage Using a Shared Key

The following connection string format should be used to connect to Azure Storage with a shared key:

```
DefaultEndpointsProtocol=[http|https];AccountName=myAccountName;AccountKey=myAccountKey
```

`myAccountName` should be replaced with the name of your storage account, and `myAccountKey` should be replaced with one of your two account access keys.

**NOTE**

When using shared key authentication, your account name and account key will be distributed to each person that uses your application, which will provide full read/write access to the storage account. Therefore, use shared key authentication for testing purposes only, and never distribute keys to other users.

## Connecting to Azure Storage using a Shared Access Signature

The following connection string format should be used to connect to Azure Storage with an SAS:

```
BlobEndpoint=myBlobEndpoint;SharedAccessSignature=mySharedAccessSignature
```

`myBlobEndpoint` should be replaced with the URL of your blob endpoint, and `mySharedAccessSignature` should be replaced with your SAS. The SAS provides the protocol, the service endpoint, and the credentials to access the resource.

**NOTE**

SAS authentication is recommended for production applications. However, in a production application the SAS should be retrieved from a backend service on-demand, rather than being bundled with the application.

For more information about Shared Access Signatures, see [Using Shared Access Signatures \(SAS\)](#).

## Creating a Container

The `GetContainer` method is used to retrieve a reference to a named container, which can then be used to retrieve blobs from the container or to add blobs to the container. The following code example shows the `GetContainer` method:

```
static CloudBlobContainer GetContainer(ContainerType containerType)
{
    var account = CloudStorageAccount.Parse(Constants.StorageConnection);
    var client = account.CreateCloudBlobClient();
    return client.GetContainerReference(containerType.ToString().ToLower());
}
```

The `CloudStorageAccount.Parse` method parses a connection string and returns a `CloudStorageAccount` instance that represents the storage account. A `CloudBlobClient` instance, which is used to retrieve containers and blobs, is then created by the `CreateCloudBlobClient` method. The `GetContainerReference` method retrieves the specified container as a `CloudBlobContainer` instance, before it's returned to the calling method. In this example, the container name is the `ContainerType` enumeration value, converted to a lowercase string.

#### NOTE

Container names must be lowercase, and must start with a letter or number. In addition, they can only contain letters, numbers, and the dash character, and must be between 3 and 63 characters long.

The `GetContainer` method is invoked as follows:

```
var container = GetContainer(containerType);
```

The `CloudBlobContainer` instance can then be used to create a container if it doesn't already exist:

```
await container.CreateIfNotExistsAsync();
```

By default, a newly created container is private. This means that a storage access key must be specified to retrieve blobs from the container. For information about making blobs within a container public, see [Create a container](#).

## Uploading Data to a Container

The `UploadFileAsync` method is used to upload a stream of byte data to blob storage, and is shown in the following code example:

```
public static async Task<string> UploadFileAsync(ContainerType containerType, Stream stream)
{
    var container = GetContainer(containerType);
    await container.CreateIfNotExistsAsync();

    var name = Guid.NewGuid().ToString();
    var fileBlob = container.GetBlockBlobReference(name);
    await fileBlob.UploadFromStreamAsync(stream);

    return name;
}
```

After retrieving a container reference, the method creates the container if it doesn't already exist. A new `Guid` is then created to act as a unique blob name, and a blob block reference is retrieved as an `CloudBlockBlob` instance. The stream of data is then uploaded to the blob using the `UploadFromStreamAsync` method, which creates the blob if it doesn't already exist, or overwrites it if it does exist.

Before a file can be uploaded to blob storage using this method, it must first be converted to a byte stream. This is demonstrated in the following code example:

```
var byteData = Encoding.UTF8.GetBytes(text);
uploadedFilename = await AzureStorage.UploadFileAsync(ContainerType.Text, new MemoryStream(byteData));
```

The `text` data is converted to a byte array, which is then wrapped as a stream that's passed to the `UploadFileAsync` method.

## Downloading Data from a Container

The `GetFileAsync` method is used to download blob data from Azure Storage, and is shown in the following code example:

```
public static async Task<byte[]> GetFileAsync(ContainerType containerType, string name)
{
    var container = GetContainer(containerType);

    var blob = container.GetBlobReference(name);
    if (await blob.ExistsAsync())
    {
        await blob.FetchAttributesAsync();
        byte[] blobBytes = new byte[blob.Properties.Length];

        await blob.DownloadToByteArrayAsync(blobBytes, 0);
        return blobBytes;
    }
    return null;
}
```

After retrieving a container reference, the method retrieves a blob reference for the stored data. If the blob exists, its properties are retrieved by the `FetchAttributesAsync` method. A byte array of the correct size is created, and the blob is downloaded as an array of bytes that gets returned to the calling method.

After downloading the blob byte data, it must be converted to its original representation. This is demonstrated in the following code example:

```
var byteData = await AzureStorage.GetFileAsync(ContainerType.Text, uploadedFilename);
string text = Encoding.UTF8.GetString(byteData);
```

The array of bytes is retrieved from Azure Storage by the `GetFileAsync` method, before it's converted back to a UTF8 encoded string.

## Listing Data in a Container

The `GetFilesListAsync` method is used to retrieve a list of blobs stored in a container, and is shown in the following code example:

```

public static async Task<IList<string>> GetFilesListAsync(ContainerType containerType)
{
    var container = GetContainer(containerType);

    var allBlobsList = new List<string>();
    BlobContinuationToken token = null;

    do
    {
        var result = await container.ListBlobsSegmentedAsync(token);
        if (result.Results.Count() > 0)
        {
            var blobs = result.Results.Cast<CloudBlockBlob>().Select(b => b.Name);
            allBlobsList.AddRange(blobs);
        }
        token = result.ContinuationToken;
    } while (token != null);

    return allBlobsList;
}

```

After retrieving a container reference, the method uses the container's `ListBlobsSegmentedAsync` method to retrieve references to the blobs within the container. The results returned by the `ListBlobsSegmentedAsync` method are enumerated while the `BlobContinuationToken` instance is not `null`. Each blob is cast from the returned `IListBlobItem` to a `CloudBlockBlob` in order access the `Name` property of the blob, before its value is added to the `allBlobsList` collection. Once the `BlobContinuationToken` instance is `null`, the last blob name has been returned, and execution exits the loop.

## Deleting Data from a Container

The `DeleteFileAsync` method is used to delete a blob from a container, and is shown in the following code example:

```

public static async Task<bool> DeleteFileAsync(ContainerType containerType, string name)
{
    var container = GetContainer(containerType);
    var blob = container.GetBlobReference(name);
    return await blob.DeleteIfExistsAsync();
}

```

After retrieving a container reference, the method retrieves a blob reference for the specified blob. The blob is then deleted with the `DeleteIfExistsAsync` method.

## Summary

This article demonstrated how to use Xamarin.Forms to store text and binary data in Azure Storage, and how to access the data. Azure Storage is a scalable cloud storage solution that can be used for storing unstructured, and structured data.

## Related Links

- [Azure Storage \(sample\)](#)
- [Introduction to Storage](#)
- [How to use Blob Storage from Xamarin](#)
- [Using Shared Access Signatures \(SAS\)](#)
- [Windows Azure Storage](#)

# Searching Data in the Cloud

4/12/2018 • 2 minutes to read • [Edit Online](#)

*Azure Search is a cloud service that provides indexing and querying capabilities for uploaded data. This removes the infrastructure requirements and search algorithm complexities traditionally associated with implementing search functionality in an application.*

## Searching Data with Azure Search

This article demonstrates how to use the Microsoft Azure Search Library to integrate Azure Search into a Xamarin.Forms application.

# Searching Data with Azure Search

6/8/2018 • 10 minutes to read • [Edit Online](#)

Azure Search is a cloud service that provides indexing and querying capabilities for uploaded data. This removes the infrastructure requirements and search algorithm complexities traditionally associated with implementing search functionality in an application. This article demonstrates how to use the Microsoft Azure Search Library to integrate Azure Search into a Xamarin.Forms application.

## Overview

Data is stored in Azure Search as indexes and documents. An *index* is a store of data that can be searched by the Azure Search service, and is conceptually similar to a database table. A *document* is a single unit of searchable data in an index, and is conceptually similar to a database row. When uploading documents and submitting search queries to Azure Search, requests are made to a specific index in the search service.

Each request made to Azure Search must include the name of the service, and an API key. There are two types of API key:

- *Admin keys* grant full rights to all operations. This includes managing the service, creating and deleting indexes, and data sources.
- *Query keys* grant read-only access to indexes and documents, and should be used by applications that issue search requests.

The most common request to Azure Search is to execute a query. There are two types of query that can be submitted:

- A *search* query searches for one or more items in all searchable fields in an index. Search queries are built using the simplified syntax, or the Lucene query syntax. For more information, see [Simple query syntax in Azure Search](#), and [Lucene query syntax in Azure Search](#).
- A *filter* query evaluates a boolean expression over all filterable fields in an index. Filter queries are built using a subset of the OData filter language. For more information, see [OData Expression Syntax for Azure Search](#).

Search queries and filter queries can be used separately or together. When used together, the filter query is applied first to the entire index, and then the search query is performed on the results of the filter query.

Azure Search also supports retrieving suggestions based on search input. For more information, see [Suggestion Queries](#).

## Setup

The process for integrating Azure Search into a Xamarin.Forms application is as follows:

1. Create an Azure Search service. For more information, see [Create an Azure Search service using the Azure Portal](#).
2. Remove Silverlight as a target framework from the Xamarin.Forms solution Portable Class Library (PCL). This can be accomplished by changing the PCL profile to any profile that supports cross-platform development, but doesn't support Silverlight, such as profile 151 or profile 92.
3. Add the [Microsoft Azure Search Library](#) NuGet package to the PCL project in the Xamarin.Forms solution.

After performing these steps, the Microsoft Search Library API can be used to manage search indexes and data sources, upload and manage documents, and execute queries.

# Creating the Azure Search Index

An index schema must be defined that maps to the structure of the data to be searched. This can be accomplished in the Azure Portal, or programmatically using the `SearchServiceClient` class. This class manages connections to Azure Search, and can be used to create an index. The following code example demonstrates how to create an instance of this class:

```
var searchClient =
    new SearchServiceClient(Constants.SearchServiceName, new SearchCredentials(Constants.AdminApiKey));
```

The `SearchServiceClient` constructor overload takes a search service name and a `SearchCredentials` object as arguments, with the `SearchCredentials` object wrapping the *admin* key for the Azure Search service. The *admin* key is required to create an index.

## NOTE

A single `SearchServiceClient` instance should be used in an application to avoid opening too many connections to Azure Search.

An index is defined by the `Index` object, as demonstrated in the following code example:

```
static void CreateSearchIndex()
{
    var index = new Index()
    {
        Name = Constants.Index,
        Fields = new[]
        {
            new Field("id", DataType.String) { IsKey = true, IsRetrievable = true },
            new Field("name", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSortable = true,
IsSearchable = true },
            new Field("location", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSortable = true,
IsSearchable = true },
            new Field("details", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSearchable = true },
            new Field("imageUrl", DataType.String) { IsRetrievable = true }
        },
        Suggesters = new[]
        {
            new Suggester("nameSuggester", SuggesterSearchMode.AnalyzingInfixMatching, new[] { "name" })
        }
    };

    searchClient.Indexes.Create(index);
}
```

The `Index.Name` property should be set to the name of the index, and the `Index.Fields` property should be set to an array of `Field` objects. Each `Field` instance specifies a name, a type, and any properties, which specify how the field is used. These properties include:

- `IsKey` – indicates whether the field is the key of the index. Only one field in the index, of type `DataType.String`, must be designated as the key field.
- `IsFacetable` – indicates whether it's possible to perform faceted navigation on this field. The default value is `false`.
- `IsFilterable` – indicates whether the field can be used in filter queries. The default value is `false`.
- `IsRetrievable` – indicates whether the field can be retrieved in search results. The default value is `true`.

- `IsSearchable` – indicates whether the field is included in full-text searches. The default value is `false`.
- `Sortable` – indicates whether the field can be used in `OrderBy` expressions. The default value is `false`.

#### NOTE

Changing an index after it's deployed involves rebuilding and reloading the data.

An `Index` object can optionally specify a `Suggesters` property, which defines the fields in the index to be used to support auto-complete or search suggestion queries. The `Suggesters` property should be set to an array of `Suggester` objects that define the fields that are used to build the search suggestion results.

After creating the `Index` object, the index is created by calling `Indexes.Create` on the `SearchServiceClient` instance.

#### NOTE

When creating an index from an application that must be kept responsive, use the `Indexes.CreateAsync` method.

For more information, see [Create an Azure Search index using the .NET SDK](#).

## Deleting the Azure Search Index

An index can be deleted by calling `Indexes.Delete` on the `SearchServiceClient` instance:

```
searchClient.Indexes.Delete(Constants.Index);
```

## Uploading Data to the Azure Search Index

After defining the index, data can be uploaded to it using one of two models:

- **Pull model** – data is periodically ingested from Azure Cosmos DB, Azure SQL Database, Azure Blob Storage, or SQL Server hosted in an Azure Virtual Machine.
- **Push model** – data is programmatically sent to the index. This is the model adopted in this article.

A `SearchIndexClient` instance must be created to import data into the index. This can be accomplished by calling the `SearchServiceClient.Indexes.GetClient` method, as demonstrated in the following code example:

```

static void UploadDataToSearchIndex()
{
    var indexClient = searchClient.Indexes.GetClient(Constants.Index);

    var monkeyList = MonkeyData.Monkeys.Select(m => new
    {
        id = Guid.NewGuid().ToString(),
        name = m.Name,
        location = m.Location,
        details = m.Details,
        imageUrl = m.ImageUrl
    });

    var batch = IndexBatch.New(monkeyList.Select(IndexAction.Upload));
    try
    {
        indexClient.Documents.Index(batch);
    }
    catch (IndexBatchException ex)
    {
        // Sometimes when the Search service is under load, indexing will fail for some
        // documents in the batch. Compensating actions like delaying and retrying should be taken.
        // Here, the failed document keys are logged.
        Console.WriteLine("Failed to index some documents: {0}",
            string.Join(", ", ex.IndexingResults.Where(r => !r.Succeeded).Select(r => r.Key)));
    }
}

```

Data to be imported into the index is packaged as an `IndexBatch` object, which encapsulates a collection of `IndexAction` objects. Each `IndexAction` instance contains a document, and a property that tells Azure Search which action to perform on the document. In the code example above, the `IndexAction.Upload` action is specified, which results in the document being inserted into the index if it's new, or replaced if it already exists. The `IndexBatch` object is then sent to the index by calling the `Documents.Index` method on the `SearchIndexClient` object. For information about other indexing actions, see [Decide which indexing action to use](#).

#### NOTE

Only 1000 documents can be included in a single indexing request.

Note that in the code example above, the `monkeyList` collection is created as an anonymous object from a collection of `Monkey` objects. This creates data for the `id` field, and resolves the mapping of Pascal case `Monkey` property names to camel case search index field names. Alternatively, this mapping can also be accomplished by adding the `[SerializePropertyNamesAsCamelCase]` attribute to the `Monkey` class.

For more information, see [Upload data to Azure Search using the .NET SDK](#).

## Querying the Azure Search Index

A `SearchIndexClient` instance must be created to query an index. When an application executes queries, it's advisable to follow the principle of least privilege and create a `SearchIndexClient` directly, passing the `query` key as an argument. This ensures that users have read-only access to indexes and documents. This approach is demonstrated in the following code example:

```

SearchIndexClient indexClient =
    new SearchIndexClient(Constants.SearchServiceName, Constants.Index, new
    SearchCredentials(Constants.QueryApiKey));

```

The `SearchIndexClient` constructor overload takes a search service name, index name, and a `SearchCredentials` object as arguments, with the `SearchCredentials` object wrapping the *query key* for the Azure Search service.

## Search Queries

The index can be queried by calling the `Documents.SearchAsync` method on the `SearchIndexClient` instance, as demonstrated in the following code example:

```
async Task AzureSearch(string text)
{
    Monkeys.Clear();

    var searchResults = await indexClient.Documents.SearchAsync<Monkey>(text);
    foreach (SearchResult<Monkey> result in searchResults.Results)
    {
        Monkeys.Add(new Monkey
        {
            Name = result.Document.Name,
            Location = result.Document.Location,
            Details = result.Document.Details,
            ImageUrl = result.Document.ImageUrl
        });
    }
}
```

The `SearchAsync` method takes a search text argument, and an optional `SearchParameters` object that can be used to further refine the query. A search query is specified as the search text argument, while a filter query can be specified by setting the `Filter` property of the `SearchParameters` argument. The following code example demonstrates both query types:

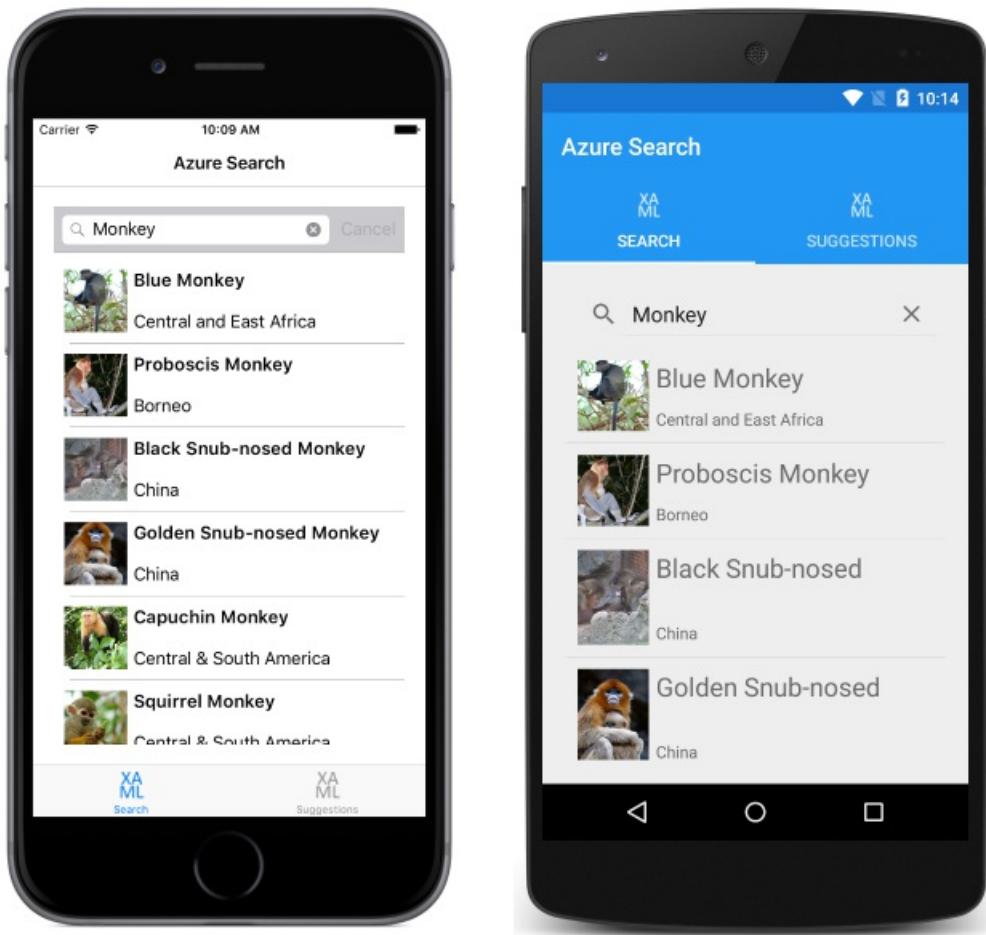
```
var parameters = new SearchParameters
{
    Filter = "location ne 'China' and location ne 'Vietnam'"
};
var searchResults = await indexClient.Documents.SearchAsync<Monkey>(text, parameters);
```

This filter query is applied to the entire index and removes documents from the results where the `location` field is not equal to China and not equal to Vietnam. After filtering, the search query is performed on the results of the filter query.

### NOTE

To filter without searching, pass `*` as the search text argument.

The `SearchAsync` method returns a `DocumentSearchResult` object that contains the query results. This object is enumerated, with each `Document` object being created as a `Monkey` object and added to the `Monkeys` `ObservableCollection` for display. The following screenshots show search query results returned from Azure Search:



For more information about searching and filtering, see [Query your Azure Search index using the .NET SDK](#).

## Suggestion Queries

Azure Search allows suggestions to be requested based on a search query, by calling the `Documents.SuggestAsync` method on the `SearchIndexClient` instance. This is demonstrated in the following code example:

```
async Task AzureSuggestions(string text)
{
    Suggestions.Clear();

    var parameters = new SuggestParameters()
    {
        UseFuzzyMatching = true,
        HighlightPreTag = "[",
        HighlightPostTag = "]",
        MinimumCoverage = 100,
        Top = 10
    };

    var suggestionResults =
        await indexClient.Documents.SuggestAsync<Monkey>(text, "nameSuggester", parameters);

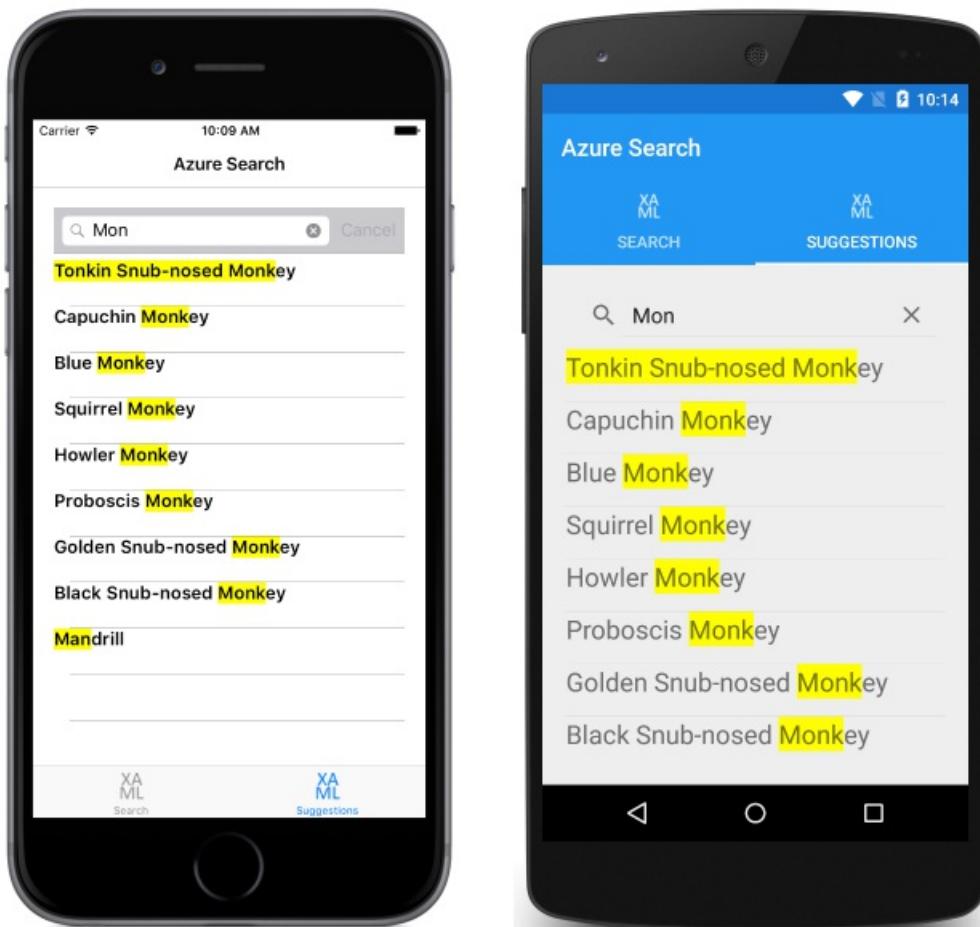
    foreach (var result in suggestionResults.Results)
    {
        Suggestions.Add(new Monkey
        {
            Name = result.Text,
            Location = result.Document.Location,
            Details = result.Document.Details,
            ImageUrl = result.Document.ImageUrl
        });
    }
}
```

The `SuggestAsync` method takes a search text argument, the name of the suggester to use (that's defined in the index), and an optional `SuggestParameters` object that can be used to further refine the query. The `SuggestParameters` instance sets the following properties:

- `UseFuzzyMatching` – when set to `true`, Azure Search will find suggestions even if there's a substituted or missing character in the search text.
- `HighlightPreTag` – the tag that is prepended to suggestion hits.
- `HighlightPostTag` – the tag that is appended to suggestion hits.
- `MinimumCoverage` – represents the percentage of the index that must be covered by a suggestion query for the query to be reported a success. The default is 80.
- `Top` – the number of suggestions to retrieve. It must be an integer between 1 and 100, with a default value of 5.

The overall effect is that the top 10 results from the index will be returned with hit highlighting, and the results will include documents that include similarly spelled search terms.

The `SuggestAsync` method returns a `DocumentSuggestResult` object that contains the query results. This object is enumerated, with each `Document` object being created as a `Monkey` object and added to the `Monkeys` `ObservableCollection` for display. The following screenshots show the suggestion results returned from Azure Search:



Note that in the sample application, the `SuggestAsync` method is only invoked when the user finishes inputting a search term. However, it can also be used to support auto-complete search queries by executing on each keypress.

## Summary

This article demonstrated how to use the Microsoft Azure Search Library to integrate Azure Search into a Xamarin.Forms application. Azure Search is a cloud service that provides indexing and querying capabilities for

uploaded data. This removes the infrastructure requirements and search algorithm complexities traditionally associated with implementing search functionality in an application.

## Related Links

- [Azure Search \(sample\)](#)
- [Azure Search Documentation](#)
- [Microsoft Azure Search Library](#)

# Serverless computing with Xamarin.Forms

8/29/2018 • 2 minutes to read • [Edit Online](#)

*Build apps with powerful back-end functionality, without the complexity of configuring and managing a server.*

## Azure Functions

Get started by building your first Azure Function that interacts with Xamarin.Forms.

# Get started with Azure Functions

11/13/2018 • 2 minutes to read • [Edit Online](#)

*Get started building your first Azure Function that interacts with Xamarin.Forms.*

- [Visual Studio](#)
- [Visual Studio for Mac](#)

## Step-by-step instructions

In addition to the video, you can follow these instructions to [build your first Function using Visual Studio 2017](#).

## Related Links

- [Azure Functions docs](#)
- [Implementing a simple Azure Function with a Xamarin.Forms client \(sample\)](#)

# Storing Data in a Document Database

4/12/2018 • 2 minutes to read • [Edit Online](#)

An Azure Cosmos DB document database is a NoSQL database that provides low latency access to JSON documents, offering a fast, highly available, scalable database service for applications that require seamless scale and global replication.

## Consuming an Azure Cosmos DB Document Database

This article explains how to use the Azure Cosmos DB .NET Standard client library to integrate an Azure Cosmos DB document database into a Xamarin.Forms application.

## Authenticating Users with an Azure Cosmos DB Document Database

This article explains how to combine access control with partitioned collections, so that a user can only access their own documents in a Xamarin.Forms application.

# Consuming an Azure Cosmos DB Document Database

11/13/2018 • 7 minutes to read • [Edit Online](#)

An Azure Cosmos DB document database is a NoSQL database that provides low latency access to JSON documents, offering a fast, highly available, scalable database service for applications that require seamless scale and global replication. This article explains how to use the Azure Cosmos DB .NET Standard client library to integrate an Azure Cosmos DB document database into a Xamarin.Forms application.

## Microsoft Azure Cosmos DB, by [Xamarin University](#)

An Azure Cosmos DB document database account can be provisioned using an Azure subscription. Each database account can have zero or more databases. A document database in Azure Cosmos DB is a logical container for document collections and users.

An Azure Cosmos DB document database may contain zero or more document collections. Each document collection can have a different performance level, allowing more throughput to be specified for frequently accessed collections, and less throughput for infrequently accessed collections.

Each document collection consists of zero or more JSON documents. Documents in a collection are schema-free, and so do not need to share the same structure or fields. As documents are added to a document collection, Cosmos DB automatically indexes them and they become available to be queried.

For development purposes, a document database can also be consumed through an emulator. Using the emulator, applications can be developed and tested locally, without creating an Azure subscription or incurring any costs. For more information about the emulator, see [Developing locally with the Azure Cosmos DB Emulator](#).

This article, and accompanying sample application, demonstrates a Todo list application where the tasks are stored in an Azure Cosmos DB document database. For more information about the sample application, see [Understanding the sample](#).

For more information about Azure Cosmos DB, see the [Azure Cosmos DB Documentation](#).

## Setup

The process for integrating an Azure Cosmos DB document database into a Xamarin.Forms application is as follows:

1. Create a Cosmos DB account. For more information, see [Create an Azure Cosmos DB account](#).
2. Add the [Azure Cosmos DB .NET Standard client library](#) NuGet package to the platform projects in the Xamarin.Forms solution.
3. Add `using` directives for the `Microsoft.Azure.Documents`, `Microsoft.Azure.Documents.Client`, and `Microsoft.Azure.Documents.Linq` namespaces to classes that will access the Cosmos DB account.

After performing these steps, the Azure Cosmos DB .NET Standard client library can be used to configure and execute requests against the document database.

#### **NOTE**

The Azure Cosmos DB .NET Standard client library can only be installed into platform projects, and not into a Portable Class Library (PCL) project. Therefore, the sample application is a Shared Access Project (SAP) to avoid code duplication. However, the `DependencyService` class can be used in a PCL project to invoke Azure Cosmos DB .NET Standard client library code contained in platform-specific projects.

## Consuming the Azure Cosmos DB account

The `DocumentClient` type encapsulates the endpoint, credentials, and connection policy used to access the Azure Cosmos DB account, and is used to configure and execute requests against the account. The following code example demonstrates how to create an instance of this class:

```
DocumentClient client = new DocumentClient(new Uri(Constants.EndpointUri), Constants.PrimaryKey);
```

The Cosmos DB Uri and primary key must be provided to the `DocumentClient` constructor. These can be obtained from the Azure Portal. For more information, see [Connect to a Azure Cosmos DB account](#).

### Creating a Database

A document database is a logical container for document collections and users, and can be created in the Azure Portal, or programmatically using the `DocumentClient.CreateDatabaseIfNotExistsAsync` method:

```
public async Task CreateDatabase(string databaseName)
{
    ...
    await client.CreateDatabaseIfNotExistsAsync(new Database
    {
        Id = databaseName
    });
    ...
}
```

The `CreateDatabaseIfNotExistsAsync` method specifies a `Database` object as an argument, with the `Database` object specifying the database name as its `Id` property. The `CreateDatabaseIfNotExistsAsync` method creates the database if it doesn't exist, or returns the database if it already exists. However, the sample application ignores any data returned by the `CreateDatabaseIfNotExistsAsync` method.

#### **NOTE**

The `CreateDatabaseIfNotExistsAsync` method returns a `Task<ResourceResponse<Database>>` object, and the status code of the response can be checked to determine whether a database was created, or an existing database was returned.

### Creating a Document Collection

A document collection is a container for JSON documents, and can be created in the Azure Portal, or programmatically using the `DocumentClient.CreateDocumentCollectionIfNotExistsAsync` method:

```

public async Task CreateDocumentCollection(string databaseName, string collectionName)
{
    ...
    // Create collection with 400 RU/s
    await client.CreateDocumentCollectionIfNotExistsAsync(
        UriFactory.CreateDatabaseUri(databaseName),
        new DocumentCollection
    {
        Id = collectionName
    },
    new RequestOptions
    {
        OfferThroughput = 400
    });
    ...
}

```

The `CreateDocumentCollectionIfNotExistsAsync` method requires two compulsory arguments – a database name specified as a `Uri`, and a `DocumentCollection` object. The `DocumentCollection` object represents a document collection whose name is specified with the `Id` property. The `CreateDocumentCollectionIfNotExistsAsync` method creates the document collection if it doesn't exist, or returns the document collection if it already exists. However, the sample application ignores any data returned by the `CreateDocumentCollectionIfNotExistsAsync` method.

#### **NOTE**

The `CreateDocumentCollectionIfNotExistsAsync` method returns a `Task<ResourceResponse<DocumentCollection>>` object, and the status code of the response can be checked to determine whether a document collection was created, or an existing document collection was returned.

Optionally, the `CreateDocumentCollectionIfNotExistsAsync` method can also specify a `RequestOptions` object, which encapsulates options that can be specified for requests issued to the Cosmos DB account. The `RequestOptions.OfferThroughput` property is used to define the performance level of the document collection, and in the sample application, is set to 400 request units per second. This value should be increased or decreased depending on whether the collection will be frequently or infrequently accessed.

#### **IMPORTANT**

Note that the `CreateDocumentCollectionIfNotExistsAsync` method will create a new collection with a reserved throughput, which has pricing implications.

## **Retrieving Document Collection Documents**

The contents of a document collection can be retrieved by creating and executing a document query. A document query is created with the `DocumentClient.CreateDocumentQuery` method:

```

public async Task<List<TodoItem>> GetTodoItemsAsync()
{
    ...
    var query = client.CreateDocumentQuery<TodoItem>(collectionLink)
        .AsDocumentQuery();
    while (query.HasMoreResults)
    {
        Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
    }
    ...
}

```

This query asynchronously retrieves all the documents from the specified collection, and places the documents in a `List<TodoItem>` collection for display.

The `CreateDocumentQuery<T>` method specifies a `Uri` argument that represents the collection that should be queried for documents. In this example, the `collectionLink` variable is a class-level field that specifies the `Uri` that represents the document collection to retrieve documents from:

```
Uri collectionLink = UriFactory.CreateDocumentCollectionUri(Constants.DatabaseName, Constants.CollectionName);
```

The `CreateDocumentQuery<T>` method creates a query that is executed synchronously, and returns an `IQueryable<T>` object. However, the `AsDocumentQuery` method converts the `IQueryable<T>` object to an `IDocumentQuery<T>` object which can be executed asynchronously. The asynchronous query is executed with the `IDocumentQuery<T>.ExecuteNextAsync` method, which retrieves the next page of results from the document database, with the `IDocumentQuery<T>.HasMoreResults` property indicating whether there are additional results to be returned from the query.

Documents can be filtered server side by including a `Where` clause in the query, which applies a filtering predicate to the query against the document collection:

```
var query = client.CreateDocumentQuery<TodoItem>(collectionLink)
    .Where(f => f.Done != true)
    .AsDocumentQuery();
```

This query retrieves all documents from the collection whose `Done` property is equal to `false`.

## Inserting a Document into a Document Collection

Documents are user defined JSON content, and can be inserted into a document collection with the `DocumentClient.CreateDocumentAsync` method:

```
public async Task SaveTodoItemAsync(TodoItem item, bool isNewItem = false)
{
    ...
    await client.CreateDocumentAsync(collectionLink, item);
    ...
}
```

The `CreateDocumentAsync` method specifies a `Uri` argument that represents the collection the document should be inserted into, and an `object` argument that represents the document to be inserted.

## Replacing a Document in a Document Collection

Documents can be replaced in a document collection with the `DocumentClient.ReplaceDocumentAsync` method:

```
public async Task SaveTodoItemAsync(TodoItem item, bool isNewItem = false)
{
    ...
    await client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
        Constants.CollectionName, item.Id), item);
    ...
}
```

The `ReplaceDocumentAsync` method specifies a `Uri` argument that represents the document in the collection that should be replaced, and an `object` argument that represents the updated document data.

## Deleting a Document from a Document Collection

A document can be deleted from a document collection with the `DocumentClient.DeleteDocumentAsync` method:

```
public async Task DeleteTodoItemAsync(string id)
{
    ...
    await client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
    Constants.CollectionName, id));
    ...
}
```

The `DeleteDocumentAsync` method specifies a `Uri` argument that represents the document in the collection that should be deleted.

## Deleting a Document Collection

A document collection can be deleted from a database with the `DocumentClient.DeleteDocumentCollectionAsync` method:

```
await client.DeleteDocumentCollectionAsync(collectionLink);
```

The `DeleteDocumentCollectionAsync` method specifies a `Uri` argument that represents the document collection to be deleted. Note that invoking this method will also delete the documents stored in the collection.

## Deleting a Database

A database can be deleted from a Cosmos DB database account with the `DocumentClient.DeleteDatabaseAsync` method:

```
await client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri(Constants.DatabaseName));
```

The `DeleteDatabaseAsync` method specifies a `Uri` argument that represents the database to be deleted. Note that invoking this method will also delete the document collections stored in the database, and the documents stored in the document collections.

## Summary

This article explained how to use the Azure Cosmos DB .NET Standard client library to integrate an Azure Cosmos DB document database into a Xamarin.Forms application. An Azure Cosmos DB document database is a NoSQL database that provides low latency access to JSON documents, offering a fast, highly available, scalable database service for applications that require seamless scale and global replication.

## Related Links

- [Todo Azure Cosmos DB \(sample\)](#)
- [Azure Cosmos DB Documentation](#)
- [Azure Cosmos DB .NET Standard client library](#)
- [Azure Cosmos DB API](#)

# Authenticating Users with an Azure Cosmos DB Document Database

6/8/2018 • 10 minutes to read • [Edit Online](#)

Azure Cosmos DB document databases support partitioned collections, which can span multiple servers and partitions, while supporting unlimited storage and throughput. This article explains how to combine access control with partitioned collections, so that a user can only access their own documents in a Xamarin.Forms application.

## Overview

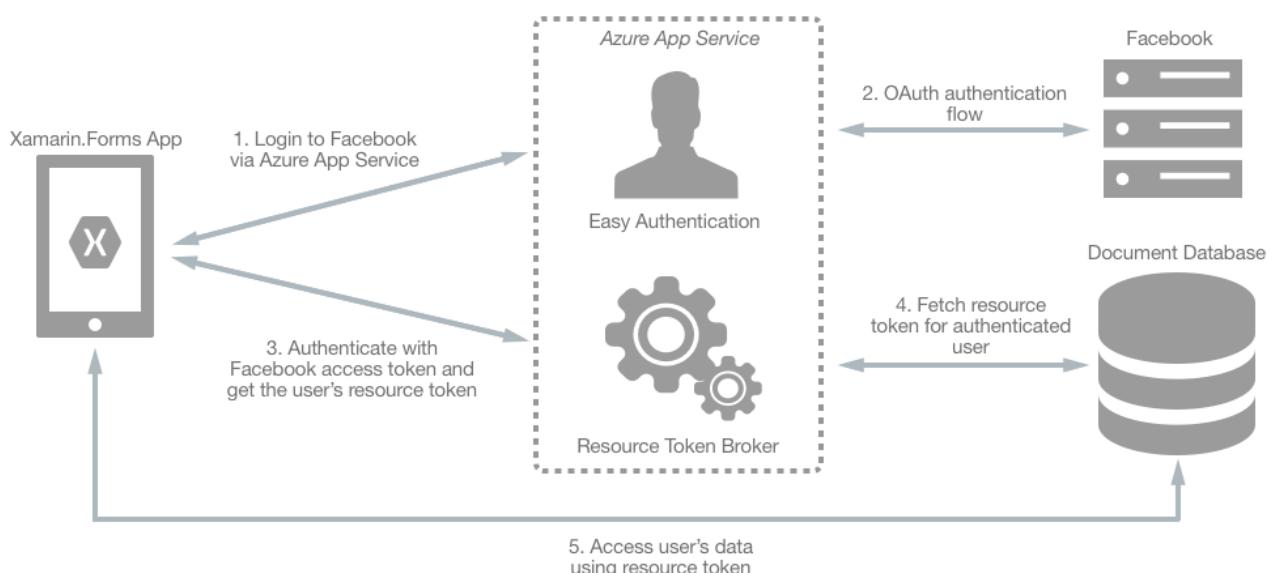
A partition key must be specified when creating a partitioned collection, and documents with the same partition key will be stored in the same partition. Therefore, specifying the user's identity as a partition key will result in a partitioned collection that will only store documents for that user. This also ensures that the Azure Cosmos DB document database will scale as the number of users and items increase.

Access must be granted to any collection, and the SQL API access control model defines two types of access constructs:

- **Master keys** enable full administrative access to all resources within a Cosmos DB account, and are created when a Cosmos DB account is created.
- **Resource tokens** capture the relationship between the user of a database and the permission the user has for a specific Cosmos DB resource, such as a collection or a document.

Exposing a master key opens a Cosmos DB account to the possibility of malicious or negligent use. However, Azure Cosmos DB resource tokens provide a safe mechanism for allowing clients to read, write, and delete specific resources in an Azure Cosmos DB account according to the granted permissions.

A typical approach to requesting, generating, and delivering resource tokens to a mobile application is to use a resource token broker. The following diagram shows a high-level overview of how the sample application uses a resource token broker to manage access to the document database data:



The resource token broker is a mid-tier Web API service, hosted in Azure App Service, which possesses the master key of the Cosmos DB account. The sample application uses the resource token broker to manage access to the document database data as follows:

1. On login, the Xamarin.Forms application contacts Azure App Service to initiate an authentication flow.
2. Azure App Service performs an OAuth authentication flow with Facebook. After the authentication flow completes, the Xamarin.Forms application receives an access token.
3. The Xamarin.Forms application uses the access token to request a resource token from the resource token broker.
4. The resource token broker uses the access token to request the user's identity from Facebook. The user's identity is then used to request a resource token from Cosmos DB, which is used to grant read/write access to the authenticated user's partitioned collection.
5. The Xamarin.Forms application uses the resource token to directly access Cosmos DB resources with the permissions defined by the resource token.

#### NOTE

When the resource token expires, subsequent document database requests will receive a 401 unauthorized exception. At this point, Xamarin.Forms applications should re-establish the identity and request a new resource token.

For more information about Cosmos DB partitioning, see [How to partition and scale in Azure Cosmos DB](#). For more information about Cosmos DB access control, see [Securing access to Cosmos DB data](#) and [Access control in the SQL API](#).

## Setup

The process for integrating the resource token broker into a Xamarin.Forms application is as follows:

1. Create a Cosmos DB account that will use access control. For more information, see [Cosmos DB Configuration](#).
2. Create an Azure App Service to host the resource token broker. For more information, see [Azure App Service Configuration](#).
3. Create a Facebook app to perform authentication. For more information, see [Facebook App Configuration](#).
4. Configure the Azure App Service to perform easy authentication with Facebook. For more information, see [Azure App Service Authentication Configuration](#).
5. Configure the Xamarin.Forms sample application to communicate with Azure App Service and Cosmos DB. For more information, see [Xamarin.Forms Application Configuration](#).

### Azure Cosmos DB Configuration

The process for creating a Cosmos DB account that will use access control is as follows:

1. Create a Cosmos DB account. For more information, see [Create an Azure Cosmos DB account](#).
2. In the Cosmos DB account, create a new collection named `UserItems`, specifying a partition key of `/userid`.

### Azure App Service Configuration

The process for hosting the resource token broker in Azure App Service is as follows:

1. In the Azure portal, create a new App Service web app. For more information, see [Create a web app in an App Service Environment](#).
2. In the Azure portal, open the App Settings blade for the web app, and add the following settings:
  - `accountUrl` – the value should be the Cosmos DB account URL from the Keys blade of the Cosmos DB account.
  - `accountKey` – the value should be the Cosmos DB master key (primary or secondary) from the Keys blade of the Cosmos DB account.
  - `databaseId` – the value should be the name of the Cosmos DB database.
  - `collectionId` – the value should be the name of the Cosmos DB collection (in this case, `UserItems`).

- `hostUrl` – the value should be the URL of the web app from the Overview blade of the App Service account.

The following screenshot demonstrates this configuration:

App settings	
WEBSITE_NODE_DEFAULT_VERSION	6.9.1
accountUrl	<a href="https://localhost:44380/documents/account">https://localhost:44380/documents/account</a>
accountKey	00000000-0000-0000-0000-000000000000
databaseId	TodoList
collectionId	UserItems
hostURL	<a href="https://localhost:44380">https://localhost:44380</a>

3. Publish the resource token broker solution to the Azure App Service web app.

## Facebook App Configuration

The process for creating a Facebook app to perform authentication is as follows:

1. Create a Facebook app. For more information, see [Register and Configure an App](#) on the Facebook Developer Center.
  2. Add the Facebook Login product to the app. For more information, see [Add Facebook Login to Your App or Website](#) on the Facebook Developer Center.
  3. Configure Facebook Login as follows:
    - Enable Client OAuth Login.
    - Enable Web OAuth Login.
    - Set the Valid OAuth redirect URI to the URI of the App Service web app, with `/.auth/login/facebook/callback` appended.

The following screenshot demonstrates this configuration:

## Client OAuth Settings

**Client OAuth Login**

Yes  No

Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]

**Web OAuth Login**

Yes  No

Enables web based OAuth client login for building custom login flows. [?]

**Force Web OAuth Reauthentication**

No

When on, prompts people to enter their Facebook password in order to log in on the web. [?]

**Embedded Browser OAuth Login**

No

Enables browser control redirect uri for OAuth client login. [?]

**Valid OAuth redirect URIs**

.auth/login/facebook/callback

**Login from Devices**

No

Enables the OAuth client login flow for devices like a smart TV [?]

For more information, see [Register your application with Facebook](#).

## Azure App Service Authentication Configuration

The process for configuring App Service easy authentication is as follows:

1. In the Azure Portal, navigate to the App Service web app.
  2. In the Azure Portal, open the Authentication / Authorization blade and perform the following configuration:

- App Service Authentication should be turned on.
- The action to take when a request is not authenticated should be set to **Login in with Facebook**.

The following screenshot demonstrates this configuration:

Authentication / Authorization

**App Service Authentication**

Off **On**

Action to take when request is not authenticated

Log in with Facebook

**Authentication Providers**

- Azure Active Directory  
Not Configured
- Facebook**  
Configured
- Google  
Not Configured
- Twitter  
Not Configured
- Microsoft Account  
Not Configured

**Advanced Settings**

Token Store Off **On**

The App Service web app should also be configured to communicate with the Facebook app to enable the authentication flow. This can be accomplished by selecting the Facebook identity provider, and entering the **App ID** and **App Secret** values from the Facebook app settings on the Facebook Developer Center. For more information, see [Add Facebook information to your application](#).

## Xamarin.Forms Application Configuration

The process for configuring the Xamarin.Forms sample application is as follows:

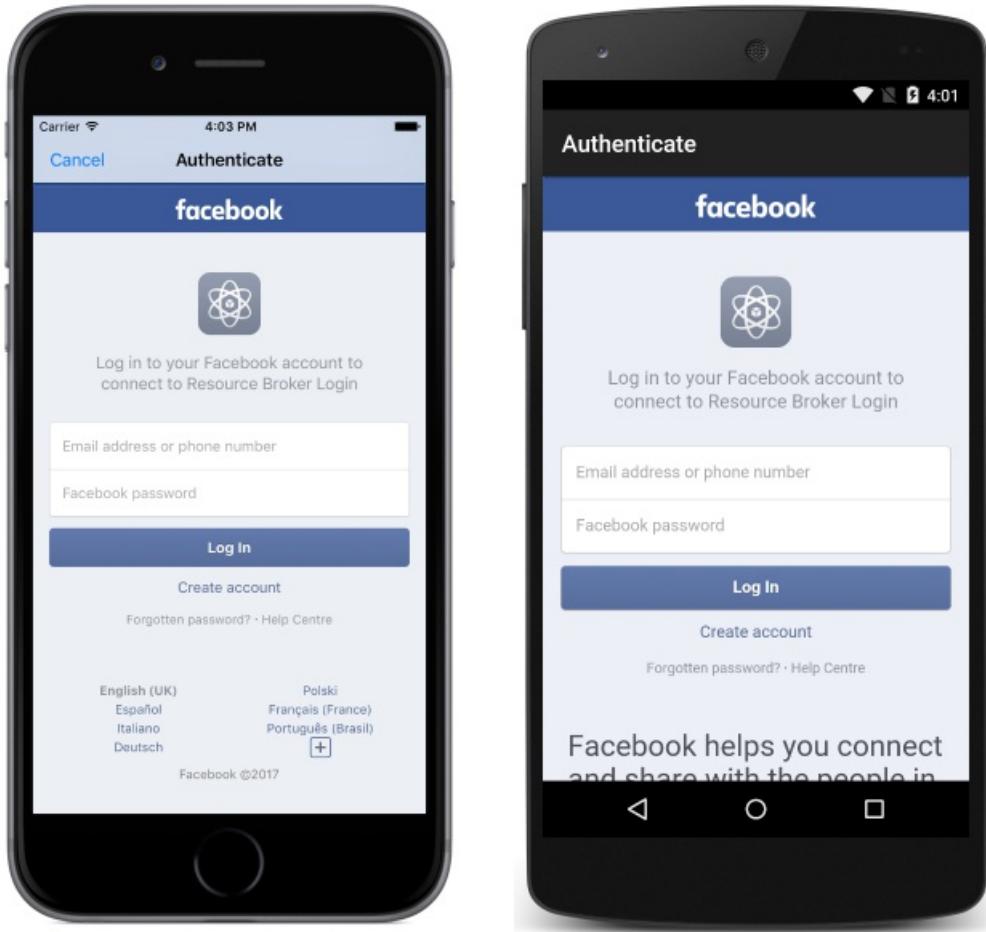
1. Open the Xamarin.Forms solution.
2. Open `Constants.cs` and update the values of the following constants:
  - `EndpointUri` – the value should be the Cosmos DB account URL from the Keys blade of the Cosmos DB account.
  - `DatabaseName` – the value should be the name of the document database.
  - `CollectionName` – the value should be the name of the document database collection (in this case, `UserItems`).
  - `ResourceTokenBrokerUrl` – the value should be the URL of the resource token broker web app from the Overview blade of the App Service account.

## Initiating Login

The sample application initiates the login process by using Xamarin.Auth to redirect a browser to an identity provider URL, as demonstrated in the following example code:

```
var auth = new Xamarin.Auth.WebRedirectAuthenticator(
  new Uri(Constants.ResourceTokenBrokerUrl + ".auth/login/facebook"),
  new Uri(Constants.ResourceTokenBrokerUrl + ".auth/login/done"));
```

This causes an OAuth authentication flow to be initiated between Azure App Service and Facebook, which displays the Facebook login page:



The login can be cancelled by pressing the **Cancel** button on iOS or by pressing the **Back** button on Android, in which case the user remains unauthenticated and the identity provider user interface is removed from the screen.

For more information about Xamarin.Auth, see [Authenticating Users with an Identity Provider](#).

## Obtaining a Resource Token

Following successful authentication, the `WebRedirectAuthenticator.Completed` event fires. The following code example demonstrates handling this event:

```

auth.Completed += async (sender, e) =>
{
    if (e.IsAuthenticated && e.Account.Properties.ContainsKey("token"))
    {
        var easyAuthResponseJson = JsonConvert.DeserializeObject< JObject>(e.Account.Properties["token"]);
        var easyAuthToken = easyAuthResponseJson.GetValue("authenticationToken").ToString();

        // Call the ResourceBroker to get the resource token
        using (var httpClient = new HttpClient())
        {
            httpClient.DefaultRequestHeaders.Add("x-zumo-auth", easyAuthToken);
            var response = await httpClient.GetAsync(Constants.ResourceTokenBrokerUrl + "/api/resourcetoken/");
            var jsonString = await response.Content.ReadAsStringAsync();
            var tokenJson = JsonConvert.DeserializeObject< JObject>(jsonString);
            resourceToken = tokenJson.GetValue("token").ToString();
            UserId = tokenJson.GetValue("userid").ToString();

            if (!string.IsNullOrWhiteSpace(resourceToken))
            {
                client = new DocumentClient(new Uri(Constants.EndpointUri), resourceToken);
                ...
            }
            ...
        }
    }
};

```

The result of a successful authentication is an access token, which is available

`AuthenticatorCompletedEventArgs.Account` property. The access token is extracted and used in a GET request to the resource token broker's `resourcetoken` API.

The `resourcetoken` API uses the access token to request the user's identity from Facebook, which in turn is used to request a resource token from Cosmos DB. If a valid permission document already exists for the user in the document database, it's retrieved and a JSON document containing the resource token is returned to the Xamarin.Forms application. If a valid permission document doesn't exist for the user, a user and permission is created in the document database, and the resource token is extracted from the permission document and returned to the Xamarin.Forms application in a JSON document.

#### NOTE

A document database user is a resource associated with a document database, and each database may contain zero or more users. A document database permission is a resource associated with a document database user, and each user may contain zero or more permissions. A permission resource provides access to a security token that the user requires when attempting to access a resource such as a document.

If the `resourcetoken` API successfully completes, it will send HTTP status code 200 (OK) in the response, along with a JSON document containing the resource token. The following JSON data shows a typical successful response message:

```
{
    "id": "John Smithpermission",
    "token": "type=resource&ver=1&sig=zx6k2zzxqktzvuzuku4b7y==;a74aukk99qtwk8v5rxfrfz7ay7zzqfkbfkremrhtaapvavw2mrvia4umbi/7iiwkrrq+buqqrzkaq4pp15y6bki1u//zf7p9x/aefbvqvq3tjjqiffurfx+vexa1xarxkkv9rbua9ypfzr47xpp5vmxuvzbekkwq6txme0xxxbjhzaxbkvzaji+iru3xqjp05amvq1r1q2k+qrarurhmjzah/ha0evixazkve2xk1zu9u/jpyf1xrwbkxqpzebvqwma+hyyazemr6qx9uz9be==",
    "expires": 4035948,
    "userid": "John Smith"
}
```

The `WebRedirectAuthenticator.Completed` event handler reads the response from the `resourcetoken` API and extracts the resource token and the user id. The resource token is then passed as an argument to the `DocumentClient` constructor, which encapsulates the endpoint, credentials, and connection policy used to access Cosmos DB, and is used to configure and execute requests against Cosmos DB. The resource token is sent with each request to directly access a resource, and indicates that read/write access to the authenticated users' partitioned collection is granted.

## Retrieving Documents

Retrieving documents that only belong to the authenticated user can be achieved by creating a document query that includes the user's id as a partition key, and is demonstrated in the following code example:

```
var query = client.CreateDocumentQuery<TodoItem>(collectionLink,
    new FeedOptions
    {
        MaxItemCount = -1,
        PartitionKey = new PartitionKey(UserId)
    })
    .Where(item => !item.Id.Contains("permission"))
    .AsDocumentQuery();
while (query.HasMoreResults)
{
    Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
}
```

The query asynchronously retrieves all the documents belonging to the authenticated user, from the specified collection, and places them in a `List<TodoItem>` collection for display.

The `CreateDocumentQuery<T>` method specifies a `Uri` argument that represents the collection that should be queried for documents, and a `FeedOptions` object. The `FeedOptions` object specifies that an unlimited number of items can be returned by the query, and the user's id as a partition key. This ensures that only documents in the user's partitioned collection are returned in the result.

### NOTE

Note that permission documents, which are created by the resource token broker, are stored in the same document collection as the documents created by the Xamarin.Forms application. Therefore, the document query contains a `Where` clause that applies a filtering predicate to the query against the document collection. This clause ensures that permission documents aren't returned from the document collection.

For more information about retrieving documents from a document collection, see [Retrieving Document Collection Documents](#).

## Inserting Documents

Prior to inserting a document into a document collection, the `TodoItem.UserId` property should be updated with the value being used as the partition key, as demonstrated in the following code example:

```
item.UserId = UserId;
await client.CreateDocumentAsync(collectionLink, item);
```

This ensures that the document will be inserted into the user's partitioned collection.

For more information about inserting a document into a document collection, see [Inserting a Document into a Document Collection](#).

## Deleting Documents

The partition key value must be specified when deleting a document from a partitioned collection, as demonstrated in the following code example:

```
await client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
    Constants.CollectionName, id),
    new RequestOptions
    {
        PartitionKey = new PartitionKey(UserId)
    });
}
```

This ensures that Cosmos DB knows which partitioned collection to delete the document from.

For more information about deleting a document from a document collection, see [Deleting a Document from a Document Collection](#).

## Summary

This article explained how to combine access control with partitioned collections, so that a user can only access their own document database documents in a Xamarin.Forms application. Specifying the user's identity as a partition key ensures that a partitioned collection can only store documents for that user.

## Related Links

- [Todo Azure Cosmos DB Auth \(sample\)](#)
- [Consuming an Azure Cosmos DB Document Database](#)
- [Securing access to Azure Cosmos DB data](#)
- [Access control in the SQL API.](#)
- [How to partition and scale in Azure Cosmos DB](#)
- [Azure Cosmos DB Client Library](#)
- [Azure Cosmos DB API](#)

# Adding Intelligence with Cognitive Services

6/8/2018 • 5 minutes to read • [Edit Online](#)

*Microsoft Cognitive Services are a set of APIs, SDKs, and services available to developers to make their applications more intelligent by adding features such as facial recognition, speech recognition, and language understanding. This article provides an introduction to the sample application that demonstrates how to invoke some of the Microsoft Cognitive Service APIs.*

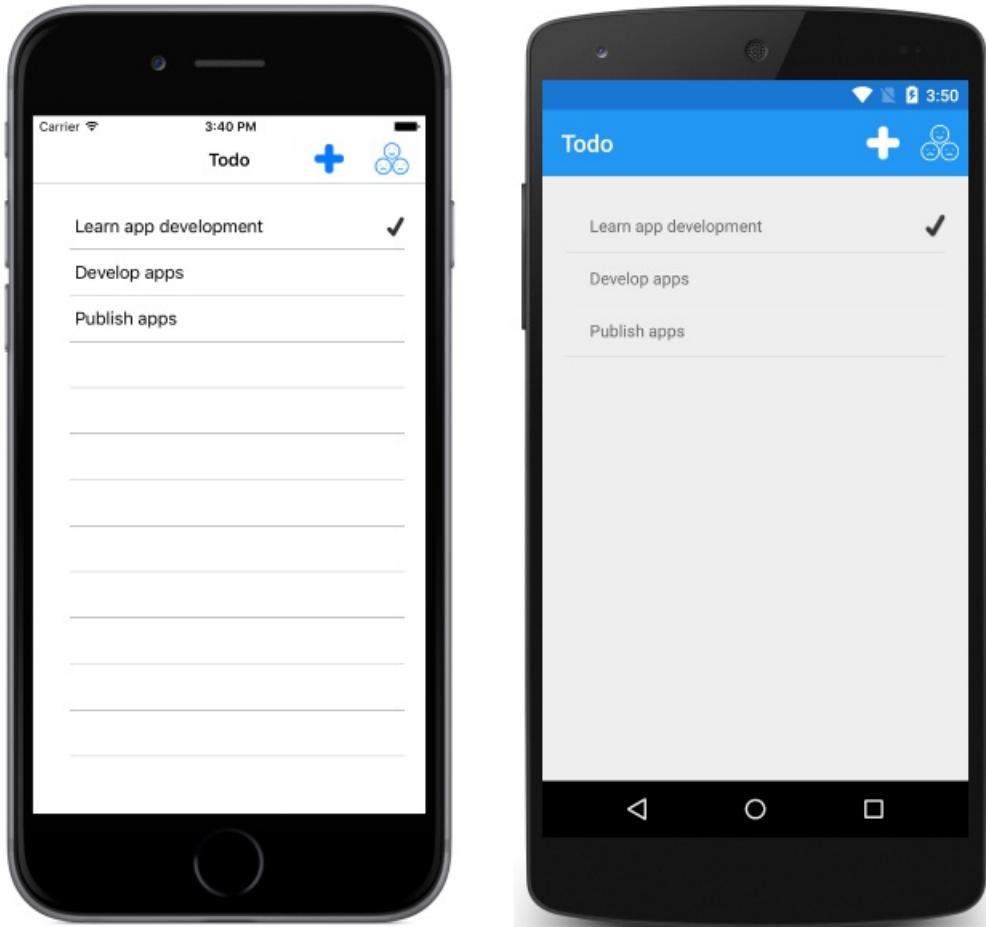
## Overview

The accompanying sample is a todo list application that provides functionality to:

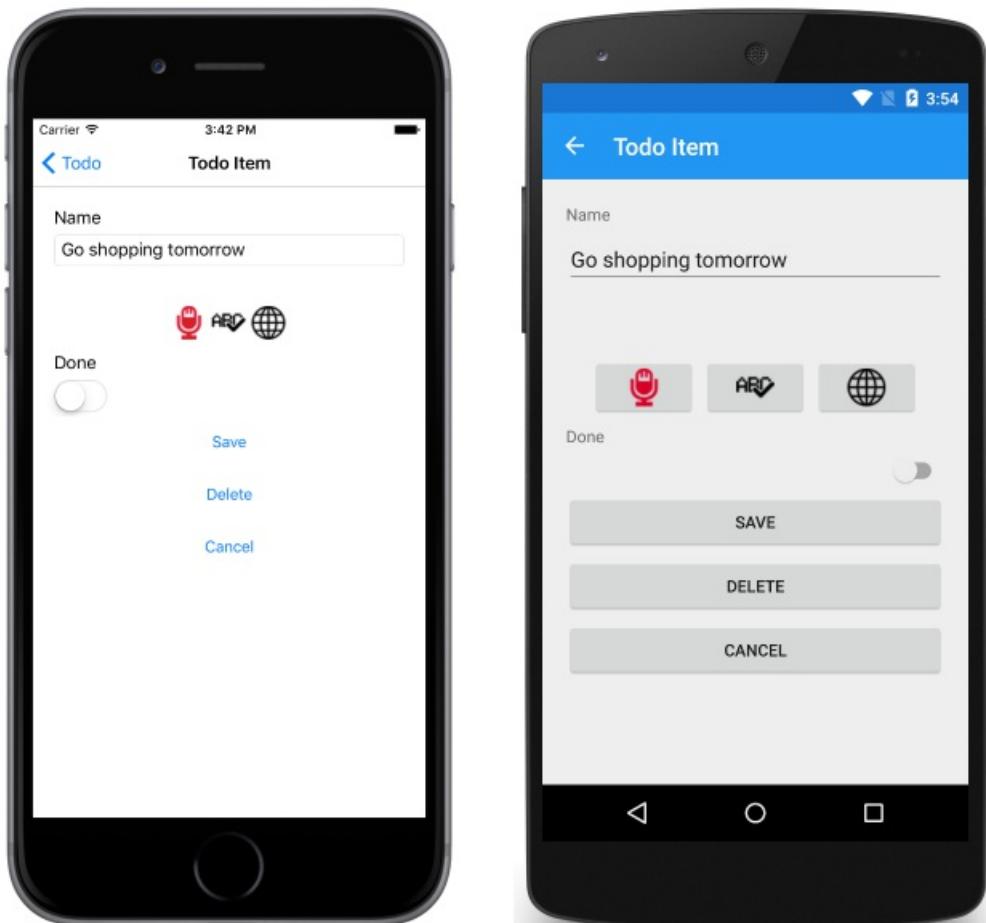
- View a list of tasks.
- Add and edit tasks through the soft keyboard, or by performing speech recognition with the Microsoft Speech API. For more information about performing speech recognition, see [Speech Recognition using the Microsoft Speech API](#).
- Spell check tasks using the Bing Spell Check API. For more information, see [Spell Checking using the Bing Spell Check API](#).
- Translate tasks from English to German using the Translator API. For more information, see [Text Translation using the Translator API](#).
- Delete tasks.
- Set a task's status to 'done'.
- Rate the application with emotion recognition, using the Face API. For more information, see [Emotion Recognition using the Face API](#).

Tasks are stored in a local SQLite database. For more information about using a local SQLite database, see [Working with a Local Database](#).

The `TodoListPage` is displayed when the application is launched. This page displays a list of any tasks stored in the local database, and allows the user to create a new task or to rate the application:

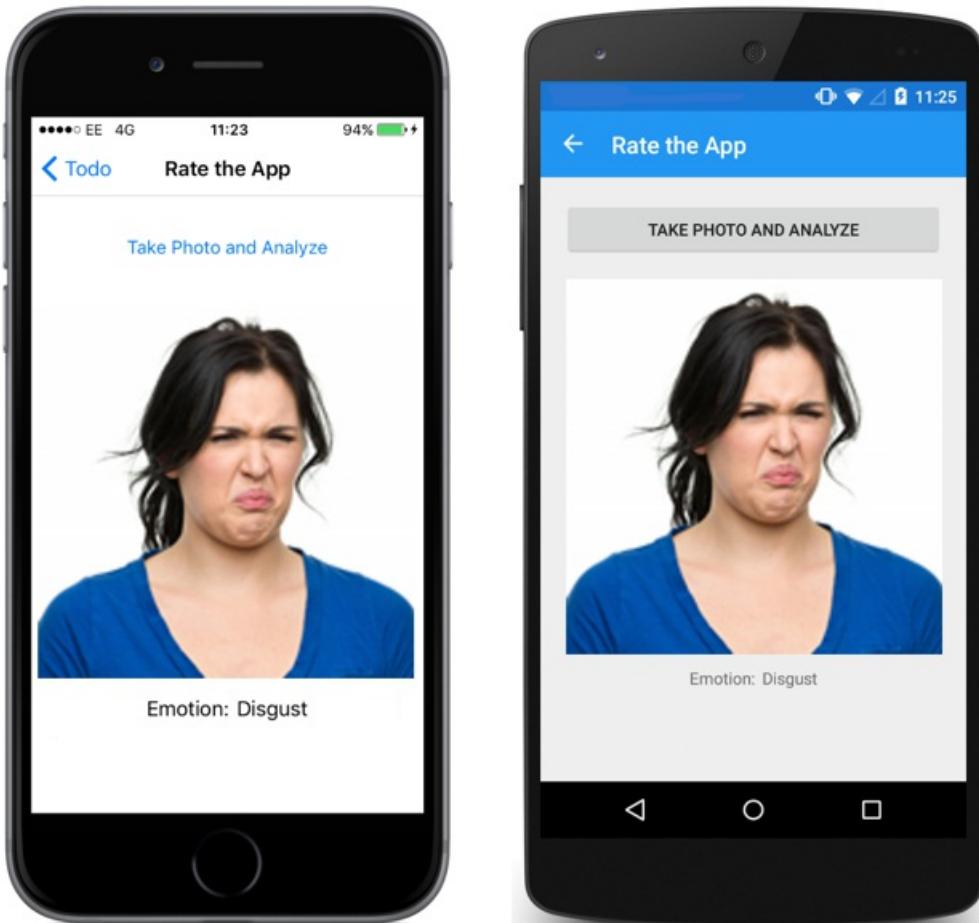


New items can be created by clicking on the + button, which navigates to the `TodoItemPage`. This page can also be navigated to by selecting a task:



The `TodoItemPage` allows tasks to be created, edited, spell-checked, translated, saved, and deleted. Speech recognition can be used to create or edit a task. This is achieved by pressing the microphone button to start recording, and by pressing the same button a second time to stop recording, which sends the recording to the Bing Speech Recognition API.

Clicking the smilies button on the `TodoListPage` navigates to the `RateAppPage`, which is used to perform emotion recognition on an image of a facial expression:



The `RateAppPage` allows the user to take a photo of their face, which is submitted to the Face API with the returned emotion being displayed.

## Understanding the Application Anatomy

The Portable Class Library (PCL) project for the sample application consists of five main folders:

FOLDER	PURPOSE
Models	Contains the data model classes for the application. This includes the <code>TodoItem</code> class, which models a single item of data used by the application. The folder also includes classes used to model JSON responses returned from different Microsoft Cognitive Service APIs.
Repositories	Contains the <code>ITodoItemRepository</code> interface and <code>TodoItemRepository</code> class that are used to perform database operations.

FOLDER	PURPOSE
Services	Contains the interfaces and classes that are used to access different Microsoft Cognitive Service APIs, along with interfaces that are used by the <code>DependencyService</code> class to locate the classes that implement the interfaces in platform projects.
Utils	Contains the <code>Timer</code> class, which is used by the <code>AuthenticationService</code> class to renew a JWT access token every 9 minutes.
Views	Contains the pages for the application.

The PCL project also contains some important files:

FILE	PURPOSE
Constants.cs	The <code>Constants</code> class, which specifies the API keys and endpoints for the Microsoft Cognitive Service APIs that are invoked. The API key constants require updating to access the different Cognitive Service APIs.
App.xaml.cs	The <code>App</code> class is responsible for instantiating both the first page that will be displayed by the application on each platform, and the <code>TodoManager</code> class that is used to invoke database operations.

## NuGet Packages

The sample application uses the following NuGet packages:

- `Newtonsoft.Json` – provides a JSON framework for .NET.
- `PCLStorage` – provides a set of cross-platform local file IO APIs.
- `sqlite-net-pcl` – provides SQLite database storage.
- `Xam.Plugin.Media` – provides cross-platform photo taking and picking APIs.

In addition, these NuGet packages also install their own dependencies.

## Modeling the Data

The sample application uses the `TodoItem` class to model the data that is displayed and stored in the local SQLite database. The following code example shows the `TodoItem` class:

```
public class TodoItem
{
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
    public string Name { get; set; }
    public bool Done { get; set; }
}
```

The `ID` property is used to uniquely identify each `TodoItem` instance, and is decorated with SQLite attributes that make the property an auto-incrementing primary key in the database.

## Invoking Database Operations

The `TodoItemRepository` class implements database operations, and an instance of the class can be accessed

through the `App.TodoManager` property. The `TodoItemRepository` class provides the following methods to invoke database operations:

- **GetAllItemsAsync** – retrieves all of the items from the local SQLite database.
- **.GetItemAsync** – retrieves a specified item from the local SQLite database.
- **SaveItemAsync** – creates or updates an item in the local SQLite database.
- **DeleteItemAsync** – deletes the specified item from the local SQLite database.

## Platform Project Implementations

The `Services` folder in the PCL project contains the `IFileHelper` and `IAudioRecorderService` interfaces that are used by the `DependencyService` class to locate the classes that implement the interfaces in platform projects.

The `IFileHelper` interface is implemented by the `FileHelper` class in each platform project. This class consists of a single method, `GetLocalFilePath`, which returns a local file path for storing the SQLite database.

The `IAudioRecorderService` interface is implemented by the `AudioRecorderService` class in each platform project. This class consists of `StartRecording`, `StopRecording`, and supporting methods, which use platform APIs to record audio from the device's microphone and store it as a wav file. On iOS, the `AudioRecorderService` uses the `AVFoundation` API to record audio. On Android, the `AudioRecordService` uses the `AudioRecord` API to record audio. On the Universal Windows Platform (UWP), the `AudioRecorderService` uses the `AudioGraph` API to record audio.

## Invoking Cognitive Services

The sample application invokes the following Microsoft Cognitive Services:

- Microsoft Speech API. For more information, see [Speech Recognition using the Microsoft Speech API](#).
- Bing Spell Check API. For more information, see [Spell Checking using the Bing Spell Check API](#).
- Translate API. For more information, see [Text Translation using the Translator API](#).
- Face API. For more information, see [Emotion Recognition using the Face API](#).

## Related Links

- [Microsoft Cognitive Services Documentation](#)
- [Todo Cognitive Services \(sample\)](#)

# Speech Recognition Using the Microsoft Speech API

11/13/2018 • 4 minutes to read • [Edit Online](#)

The Microsoft Speech API is a cloud-based API that provides algorithms to process spoken language. This article explains how to use the Microsoft Speech Recognition REST API to convert audio to text in a Xamarin.Forms application.

## Overview

The Microsoft Speech API has two components:

- A speech recognition API for converting spoken words to text. Speech recognition can be performed via a REST API, client library, or service library.
- A text to speech API for converting text into spoken words. Text to speech conversion is performed via a REST API.

This article focuses on performing speech recognition via the REST API. While the client and service libraries support returning partial results, the REST API can only return a single recognition result, without any partial results.

An API key must be obtained to use the Microsoft Speech API. This can be obtained from the Azure [portal](#). For more information, see [Create a Cognitive Services account in the Azure portal](#).

For more information about the Microsoft Speech API, see [Microsoft Speech API Documentation](#).

## Authentication

Every request made to the Microsoft Speech REST API requires a JSON Web Token (JWT) access token, which can be obtained from the cognitive services token service at

`https://api.cognitive.microsoft.com/sts/v1.0/issueToken`. A token can be obtained by making a POST request to the token service, specifying an `Ocp-Apim-Subscription-Key` header that contains the API key as its value.

The following code example shows how to request an access token from the token service:

```
public AuthenticationService(string apiKey)
{
    subscriptionKey = apiKey;
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", apiKey);
}

...
async Task<string> FetchTokenAsync(string fetchUri)
{
    UriBuilder uriBuilder = new UriBuilder(fetchUri);
    uriBuilder.Path += "/issueToken";
    var result = await httpClient.PostAsync(uriBuilder.Uri.AbsoluteUri, null);
    return await result.Content.ReadAsStringAsync();
}
```

The returned access token, which is Base64 text, has an expiry time of 10 minutes. Therefore, the sample application renews the access token every 9 minutes.

The access token must be specified in each Microsoft Speech REST API call as an `Authorization` header prefixed with the string `Bearer`, as shown in the following code example:

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);
```

Failure to pass a valid access token to the Microsoft Speech REST API will result in a 403 response error.

## Performing Speech Recognition

Speech recognition is achieved by making a POST request to the `recognition` API at <https://speech.platform.bing.com/speech/recognition/>. A single request can't contain more than 10 seconds of audio, and the total request duration can't exceed 14 seconds.

Audio content must be placed in the POST body of the request in wav format.

In the sample application, the `RecognizeSpeechAsync` method invokes the speech recognition process:

```
public async Task<SpeechResult> RecognizeSpeechAsync(string filename)
{
    ...

    // Read audio file to a stream
    var file = await PCLStorage.FileSystem.Current.LocalStorage.GetFileAsync(filename);
    var fileStream = await file.OpenAsync(PCLStorage.FileAccess.Read);

    // Send audio stream to Bing and deserialize the response
    string requestUri = GenerateRequestUri(Constants.SpeechRecognitionEndpoint);
    string accessToken = authenticationService.GetAccessToken();
    var response = await SendRequestAsync(fileStream, requestUri, accessToken, Constants.AudioContentType);
    var speechResult = JsonConvert.DeserializeObject<SpeechResult>(response);

    fileStream.Dispose();
    return speechResult;
}
```

Audio is recorded in each platform-specific project as PCM wav data, and the `RecognizeSpeechAsync` method uses the `PCLStorage` NuGet package to open the audio file as a stream. The speech recognition request URI is generated and an access token is retrieved from the token service. The speech recognition request is posted to the `recognition` API, which returns a JSON response containing the result. The JSON response is deserialized, with the result being returned to the calling method for display.

## Configuring Speech Recognition

The speech recognition process can be configured by specifying HTTP query parameters:

```
string GenerateRequestUri(string speechEndpoint)
{
    // To build a request URL, you should follow:
    // https://docs.microsoft.com/azure/cognitive-services/speech/getstarted/getstartedrest
    string requestUri = speechEndpoint;
    requestUri += @"dictation/cognitiveservices/v1?";
    requestUri += @"language=en-us";
    requestUri += @"&format=simple";
    System.Diagnostics.Debug.WriteLine(requestUri.ToString());
    return requestUri;
}
```

The main configuration performed by the `GenerateRequestUri` method is to set the locale of the audio content. For a list of the supported locales, see [Supported languages](#).

## Sending the Request

The `SendRequestAsync` method makes the POST request to the Microsoft Speech REST API and returns the

response:

```
async Task<string> SendRequestAsync(Stream fileStream, string url, string bearerToken, string contentType)
{
    if (httpClient == null)
    {
        httpClient = new HttpClient();
    }
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);

    var content = new StreamContent(fileStream);
    content.Headers.TryAddWithoutValidation("Content-Type", contentType);
    var response = await httpClient.PostAsync(url, content);
    return await response.Content.ReadAsStringAsync();
}
```

This method builds the POST request by:

- Wrapping the audio stream in a `StreamContent` instance, which provides HTTP content based on a stream.
- Setting the `Content-Type` header of the request to `audio/wav; codec="audio/pcm"; samplerate=16000`.
- Adding the access token to the `Authorization` header, prefixed with the string `Bearer`.

The POST request is then sent to `recognition` API. The response is then read and returned to the calling method.

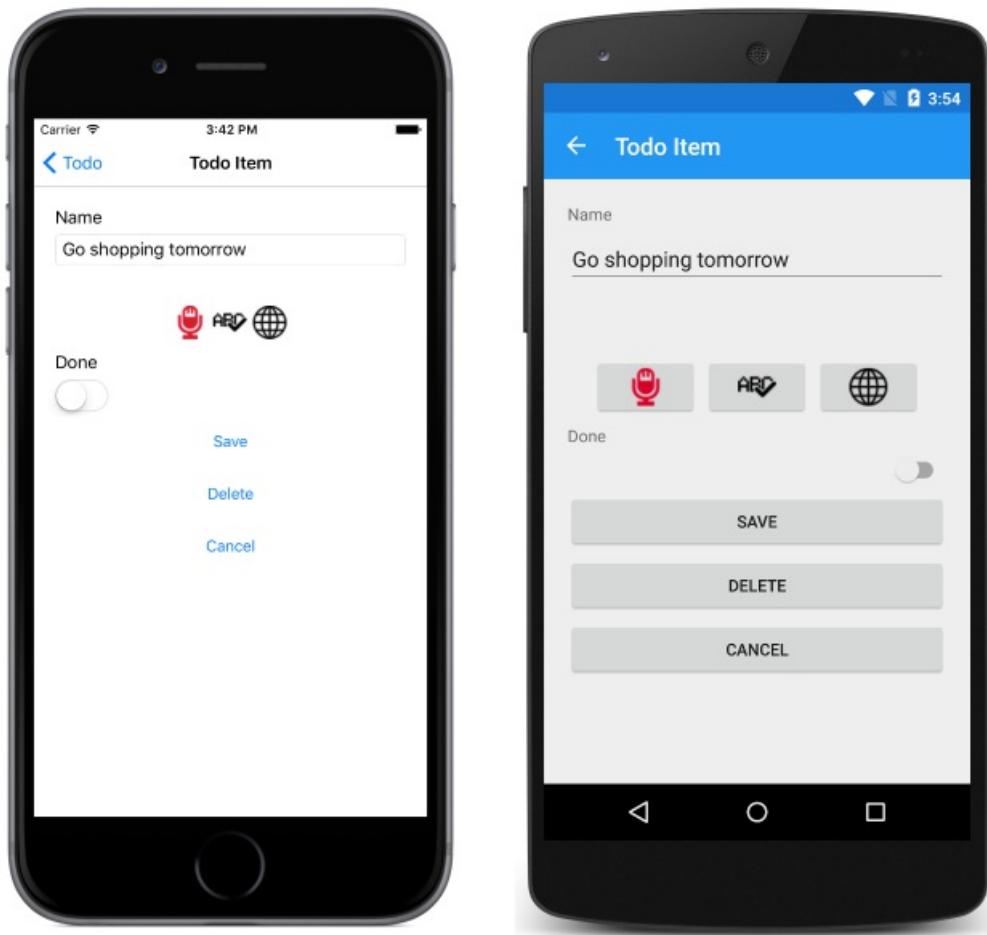
The `recognition` API will send HTTP status code 200 (OK) in the response, provided that the request is valid, which indicates that the request succeeded and that the requested information is in the response. For a list of possible error responses, see [Troubleshooting](#).

## Processing the Response

The API response is returned in JSON format, with the recognized text being contained in the `name` tag. The following JSON data shows a typical successful response message:

```
{
    "RecognitionStatus": "Success",
    "DisplayText": "Go shopping tomorrow.",
    "Offset": 16000000,
    "Duration": 17100000
}
```

In the sample application, the JSON response is deserialized into a `SpeechResult` instance, with the result being returned to the calling method for display, as shown in the following screenshots:



## Summary

This article explained how to use the Microsoft Speech REST API to convert audio to text in a Xamarin.Forms application. In addition to performing speech recognition, the Microsoft Speech API can also convert text into spoken words.

## Related Links

- [Microsoft Speech API Documentation](#).
- [Consuming a RESTful Web Service](#)
- [Todo Cognitive Services \(sample\)](#)

# Spell Checking Using the Bing Spell Check API

5/10/2018 • 4 minutes to read • [Edit Online](#)

*Bing Spell Check performs contextual spell checking for text, providing inline suggestions for misspelled words. This article explains how to use the Bing Spell Check REST API to correct spelling errors in a Xamarin.Forms application.*

## Overview

The Bing Spell Check REST API has two operating modes, and a mode must be specified when making a request to the API:

- `Spell` corrects short text (up to 9 words) without any casing changes.
- `Proof` corrects long text, provides casing corrections and basic punctuation, and suppresses aggressive corrections.

An API key must be obtained to use the Bing Spell Check API. This can be obtained at [Try Cognitive Services](#)

For a list of the languages supported by the Bing Spell Check API, see [Supported languages](#). For more information about the Bing Spell Check API, see [Bing Spell Check Documentation](#).

## Authentication

Every request made to the Bing Spell Check API requires an API key that should be specified as the value of the `Ocp-Apim-Subscription-Key` header. The following code example shows how to add the API key to the `Ocp-Apim-Subscription-Key` header of a request:

```
public BingSpellCheckService()
{
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", Constants.BingSpellCheckApiKey);
}
```

Failure to pass a valid API key to the Bing Spell Check API will result in a 401 response error.

## Performing Spell Checking

Spell checking can be achieved by making a GET or POST request to the `SpellCheck` API at <https://api.cognitive.microsoft.com/bing/v7.0/SpellCheck>. When making a GET request, the text to be spell checked is sent as a query parameter. When making a POST request, the text to be spell checked is sent in the request body. GET requests are limited to spell checking 1500 characters due to the query parameter string length limitation. Therefore, POST requests should typically be made unless short strings are being spell checked.

In the sample application, the `SpellCheckTextAsync` method invokes the spell checking process:

```
public async Task<SpellCheckResult> SpellCheckTextAsync(string text)
{
    string requestUri = GenerateRequestUri(Constants.BingSpellCheckEndpoint, text, SpellCheckMode.Spell);
    var response = await SendRequestAsync(requestUri);
    var spellCheckResults = JsonConvert.DeserializeObject<SpellCheckResult>(response);
    return spellCheckResults;
}
```

The `SpellCheckTextAsync` method generates a request URI and then sends the request to the `SpellCheck` API, which returns a JSON response containing the result. The JSON response is deserialized, with the result being returned to the calling method for display.

## Configuring Spell Checking

The spell checking process can be configured by specifying HTTP query parameters:

```
string GenerateRequestUri(string spellCheckEndpoint, string text, SpellCheckMode mode)
{
    string requestUri = spellCheckEndpoint;
    requestUri += string.Format("?text={0}", text); // text to spell check
    requestUri += string.Format("&mode={0}", mode.ToString().ToLower()); // spellcheck mode - proof or spell
    return requestUri;
}
```

This method sets the text to be spell checked, and the spell check mode.

For more information about the Bing Spell Check REST API, see [Spell Check API v7 reference](#).

## Sending the Request

The `SendRequestAsync` method makes the GET request to the Bing Spell Check REST API and returns the response:

```
async Task<string> SendRequestAsync(string url)
{
    var response = await httpClient.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}
```

This method sends the GET request to the `SpellCheck` API, with the request URL specifying the text to be translated, and the spell check mode. The response is then read and returned to the calling method.

The `SpellCheck` API will send HTTP status code 200 (OK) in the response, provided that the request is valid, which indicates that the request succeeded and that the requested information is in the response. For a list of response objects, see [Response objects](#).

## Processing the Response

The API response is returned in JSON format. The following JSON data shows the response message for the misspelled text `Go shappin tommorow`:

```
{
    "_type": "SpellCheck",
    "flaggedTokens": [
        {
            "offset": 3,
            "token": "shappin",
            "type": "UnknownToken",
            "suggestions": [
                {
                    "suggestion": "shopping",
                    "score": 1
                }
            ]
        },
        {
            "offset": 11,
            "token": "tommorow",
            "type": "UnknownToken",
            "suggestions": [
                {
                    "suggestion": "tomorrow",
                    "score": 1
                }
            ]
        }
    ],
    "correctionType": "High"
}
```

The `flaggedTokens` array contains an array of words in the text that were flagged as not being spelled correctly or are grammatically incorrect. The array will be empty if no spelling or grammar errors are found. The tags within the array are:

- `offset` – a zero-based offset from the beginning of the text string to the word that was flagged.
- `token` – the word in the text string that is not spelled correctly or is grammatically incorrect.
- `type` – the type of the error that caused the word to be flagged. There are two possible values – `RepeatedToken` and `UnknownToken`.
- `suggestions` – an array of words that will correct the spelling or grammar error. The array is made up of a `suggestion` and a `score`, which indicates the level of confidence that the suggested correction is correct.

In the sample application, the JSON response is deserialized into a `SpellCheckResult` instance, with the result being returned to the calling method for display. The following code example shows how the `SpellCheckResult` instance is processed for display:

```
var spellCheckResult = await bingSpellCheckService.SpellCheckTextAsync(TodoItem.Name);
foreach (var flaggedToken in spellCheckResult.FlaggedTokens)
{
    TodoItem.Name = TodoItem.Name.Replace(flaggedToken.Token,
    flaggedToken.Suggestions.FirstOrDefault().Suggestion);
}
```

This code iterates through the `FlaggedTokens` collection and replaces any misspelled or grammatically incorrect words in the source text with the first suggestion. The following screenshots show before and after the spell check:

# Before



iOS

Android

# After



iOS

Android

## Summary

This article explained how to use the Bing Spell Check REST API to correct spelling errors in a Xamarin.Forms application. Bing Spell Check performs contextual spell checking for text, providing inline suggestions for misspelled words.

## Related Links

- [Bing Spell Check Documentation](#)
- [Consuming a RESTful Web Service](#)
- [Todo Cognitive Services \(sample\)](#)
- [Bing Spell Check API v7 reference](#)

# Text Translation Using the Translator API

4/12/2018 • 4 minutes to read • [Edit Online](#)

The Microsoft Translator API can be used to translate speech and text through a REST API. This article explains how to use the Microsoft Translator Text API to translate text from one language to another in a Xamarin.Forms application.

## Overview

The Translator API has two components:

- A text translation REST API to translate text from one language into text of another language. The API automatically detects the language of the text that was sent before translating it.
- A speech translation REST API to transcribe speech from one language into text of another language. The API also integrates text-to-speech capabilities to speak the translated text back.

This article focuses on translating text from one language to another using the Translator Text API.

An API key must be obtained to use the Translator Text API. This can be obtained at [How to sign up for the Microsoft Translator Text API](#).

For more information about the Microsoft Translator Text API, see [Translator Text API Documentation](#).

## Authentication

Every request made to the Translator Text API requires a JSON Web Token (JWT) access token, which can be obtained from the cognitive services token service at <https://api.cognitive.microsoft.com/sts/v1.0/issueToken>. A token can be obtained by making a POST request to the token service, specifying an `Ocp-Apim-Subscription-Key` header that contains the API key as its value.

The following code example shows how to request an access token from the token service:

```
public AuthenticationService(string apiKey)
{
    subscriptionKey = apiKey;
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", apiKey);
}
...
async Task<string> FetchTokenAsync(string fetchUri)
{
    UriBuilder uriBuilder = new UriBuilder(fetchUri);
    uriBuilder.Path += "/issueToken";
    var result = await httpClient.PostAsync(uriBuilder.Uri.AbsoluteUri, null);
    return await result.Content.ReadAsStringAsync();
}
```

The returned access token, which is Base64 text, has an expiry time of 10 minutes. Therefore, the sample application renews the access token every 9 minutes.

The access token must be specified in each Translator Text API call as an `Authorization` header prefixed with the string `Bearer`, as shown in the following code example:

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);
```

For more information about the cognitive services token service, see [Authentication Token API](#).

## Performing Text Translation

Text translation can be achieved by making a GET request to the `translate` API at

`https://api.microsofttranslator.com/v2/http.svc/translate`. In the sample application, the `TranslateTextAsync` method invokes the text translation process:

```
public async Task<string> TranslateTextAsync(string text)
{
    ...
    string requestUri = GenerateRequestUri(Constants.TextTranslatorEndpoint, text, "en", "de");
    string accessToken = authenticationService.GetAccessToken();
    var response = await SendRequestAsync(requestUri, accessToken);
    var xml = XDocument.Parse(response);
    return xml.Root.Value;
}
```

The `TranslateTextAsync` method generates a request URI and retrieves an access token from the token service. The text translation request is then sent to the `translate` API, which returns an XML response containing the result. The XML response is parsed, and the translation result is returned to the calling method for display.

For more information about the Text Translation REST APIs, see [Microsoft Translator Text API](#).

### Configuring Text Translation

The text translation process can be configured by specifying HTTP query parameters:

```
string GenerateRequestUri(string endpoint, string text, string to)
{
    string requestUri = endpoint;
    requestUri += string.Format("?text={0}", Uri.EscapeUriString(text));
    requestUri += string.Format("&to={0}", to);
    return requestUri;
}
```

This method sets the text to be translated, and the language to translate the text to. For a list of the languages supported by Microsoft Translator, see [Supported languages in the Microsoft Translator Text API](#).

#### NOTE

If an application needs to know what language the text is in, the `Detect` API can be called to detect the language of the text string.

### Sending the Request

The `SendRequestAsync` method makes the GET request to the Text Translation REST API and returns the response:

```

async Task<string> SendRequestAsync(string url, string bearerToken)
{
    if (httpClient == null)
    {
        httpClient = new HttpClient();
    }
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);

    var response = await httpClient.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}

```

This method builds the GET request by adding the access token to the `Authorization` header, prefixed with the string `Bearer`. The GET request is then sent to the `translate` API, with the request URL specifying the text to be translated, and the language to translate the text to. The response is then read and returned to the calling method.

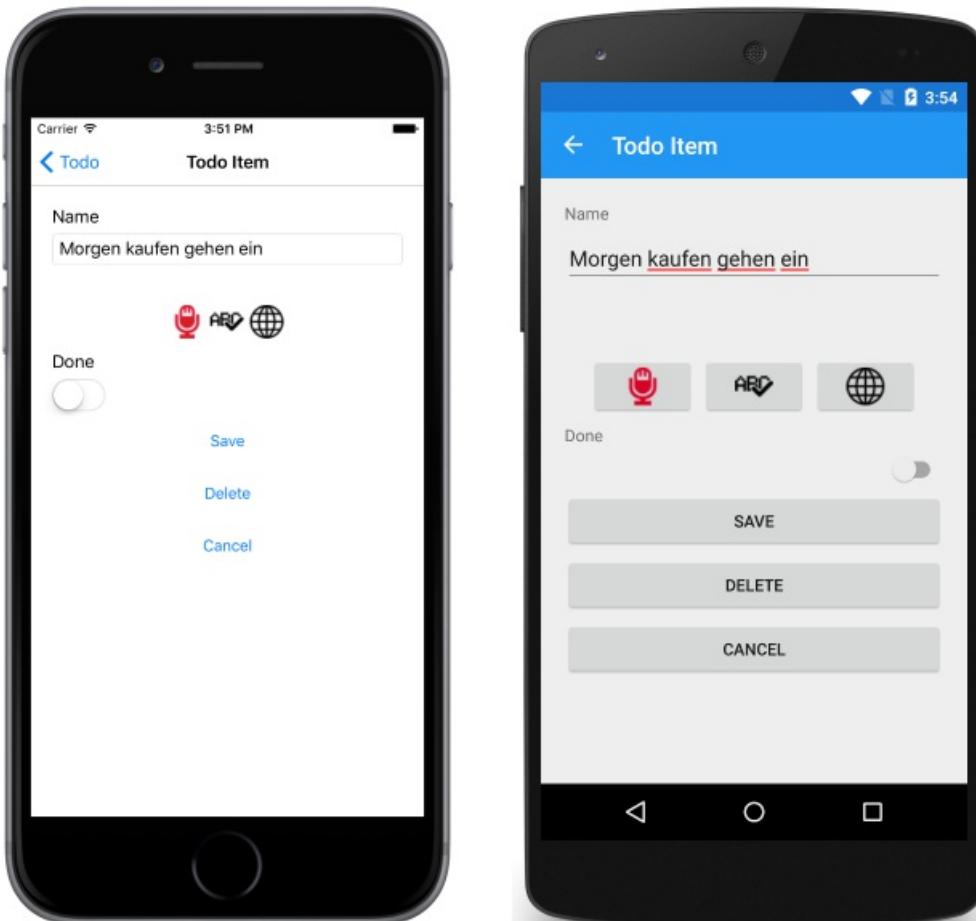
The `translate` API will send HTTP status code 200 (OK) in the response, provided that the request is valid, which indicates that the request succeeded and that the requested information is in the response. For a list of possible error responses, see Response Messages at [GET Translate](#).

## Processing the Response

The API response is returned in XML format. The following XML data shows a typical successful response message:

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Morgen kaufen gehen ein</string>
```

In the sample application, the XML response is parsed into a `XDocument` instance, with the XML root value being returned to the calling method for display as shown in the following screenshots:



## Summary

This article explained how to use the Microsoft Translator Text API to translate text from one language into text of another language in a Xamarin.Forms application. In addition to translating text, the Microsoft Translator API can also transcribe speech from one language into text of another language.

## Related Links

- [Translator Text API Documentation](#).
- [Consuming a RESTful Web Service](#)
- [Todo Cognitive Services \(sample\)](#)
- [Microsoft Translator Text API](#).

# Emotion Recognition Using the Face API

5/10/2018 • 5 minutes to read • [Edit Online](#)

The Face API takes a facial expression in an image as an input, and returns data that includes confidence levels across a set of emotions for each face in the image. This article explains how to use the Face API to recognize emotion, to rate a Xamarin.Forms application.

## Overview

The Face API can perform emotion detection to detect anger, contempt, disgust, fear, happiness, neutral, sadness, and surprise, in a facial expression. These emotions are universally and cross-culturally communicated via the same basic facial expressions. As well as returning an emotion result for a facial expression, the Face API can also return a bounding box for detected faces. Note that an API key must be obtained to use the Face API. This can be obtained at [Try Cognitive Services](#).

Emotion recognition can be performed via a client library, and via a REST API. This article focuses on performing emotion recognition via the REST API. For more information about the REST API, see [Face REST API](#).

The Face API can also be used to recognize the facial expressions of people in video, and can return a summary of their emotions. For more information, see [How to Analyze Videos in Real-time](#).

For more information about the Face API, see [Face API](#).

## Authentication

Every request made to the Face API requires an API key that should be specified as the value of the `Ocp-Apim-Subscription-Key` header. The following code example shows how to add the API key to the `Ocp-Apim-Subscription-Key` header of a request:

```
public FaceRecognitionService()
{
    _client = new HttpClient();
    _client.DefaultRequestHeaders.Add("ocp-apim-subscription-key", Constants.FaceApiKey);
}
```

Failure to pass a valid API key to the Face API will result in a 401 response error.

## Performing Emotion Recognition

Emotion recognition is performed by making a POST request containing an image to the `detect` API at [https://\[location\].api.cognitive.microsoft.com/face/v1.0](https://[location].api.cognitive.microsoft.com/face/v1.0), where `[location]` is the region you used to obtain your API key. The optional request parameters are:

- `returnFaceId` – whether to return `faceIds` of the detected faces. The default value is `true`.
- `returnFaceLandmarks` – whether to return face landmarks of the detected faces. The default value is `false`.
- `returnFaceAttributes` – whether to analyze and return one or more specified face attributes. Supported face attributes include `age`, `gender`, `headPose`, `smile`, `facialHair`, `glasses`, `emotion`, `hair`, `makeup`, `occlusion`, `accessories`, `blur`, `exposure`, and `noise`. Note that face attribute analysis has additional computational and time cost.

Image content must be placed in the body of the POST request as a URL, or binary data.

#### **NOTE**

Supported image file formats are JPEG, PNG, GIF, and BMP, and the allowed file size is from 1KB to 4MB.

In the sample application, the emotion recognition process is invoked by calling the `DetectAsync` method:

```
Face[] faces = await _faceRecognitionService.DetectAsync(photoStream, true, false, new FaceAttributeType[] { FaceAttributeType.Emotion });
```

This method call specifies the stream containing the image data, that faceIds should be returned, that face landmarks shouldn't be returned, and that the emotion of the image should be analyzed. It also specifies that the results will be returned as an array of `Face` objects. In turn, the `DetectAsync` method invokes the `detect` REST API that performs emotion recognition:

```
public async Task<Face[]> DetectAsync(Stream imageStream, bool returnFaceId, bool returnFaceLandmarks, IEnumerable<FaceAttributeType> returnFaceAttributes)
{
    var requestUrl =
        $"{Constants.FaceEndpoint}/detect?returnFaceId={returnFaceId}" +
        "&returnFaceLandmarks={returnFaceLandmarks}" +
        "&returnFaceAttributes={GetAttributeString(returnFaceAttributes)}";
    return await SendRequestAsync<Stream, Face[]>(HttpMethod.Post, requestUrl, imageStream);
}
```

This method generates a request URI and then sends the request to the `detect` API via the `SendRequestAsync` method.

#### **NOTE**

You must use the same region in your Face API calls as you used to obtain your subscription keys. For example, if you obtained your subscription keys from the `westus` region, the face detection endpoint will be

`https://westus.api.cognitive.microsoft.com/face/v1.0/detect`.

## Sending the Request

The `SendRequestAsync` method makes the POST request to the Face API and returns the result as a `Face` array:

```

async Task<TResponse> SendRequestAsync<TRequest, TResponse>(HttpMethod httpMethod, string requestUrl, TRequest
requestBody)
{
    var request = new HttpRequestMessage(httpMethod, Constants.FaceEndpoint);
    request.RequestUri = new Uri(requestUrl);
    if (requestBody != null)
    {
        if (requestBody is Stream)
        {
            request.Content = new StreamContent(requestBody as Stream);
            request.Content.Headers.ContentType = new MediaTypeHeaderValue("application/octet-stream");
        }
        else
        {
            // If the image is supplied via a URL
            request.Content = new StringContent(JsonConvert.SerializeObject(requestBody, s_settings), Encoding.UTF8,
"application/json");
        }
    }

    HttpResponseMessage responseMessage = await _client.SendAsync(request);
    if (responseMessage.IsSuccessStatusCode)
    {
        string responseContent = null;
        if (responseMessage.Content != null)
        {
            responseContent = await responseMessage.Content.ReadAsStringAsync();
        }
        if (!string.IsNullOrWhiteSpace(responseContent))
        {
            return JsonConvert.DeserializeObject<TResponse>(responseContent, s_settings);
        }
        return default(TResponse);
    }
    else
    {
        ...
    }
    return default(TResponse);
}

```

If the image is supplied via a stream, the method builds the POST request by wrapping the image stream in a `StreamContent` instance, which provides HTTP content based on a stream. Alternatively, if the image is supplied via a URL, the method builds the POST request by wrapping the URL in a `StringContent` instance, which provides HTTP content based on a string.

The POST request is then sent to `detect` API. The response is read, deserialized, and returned to the calling method.

The `detect` API will send HTTP status code 200 (OK) in the response, provided that the request is valid, which indicates that the request succeeded and that the requested information is in the response. For a list of possible error responses, see [Face REST API](#).

## Processing the Response

The API response is returned in JSON format. The following JSON data shows a typical successful response message that supplies the data requested by the sample application:

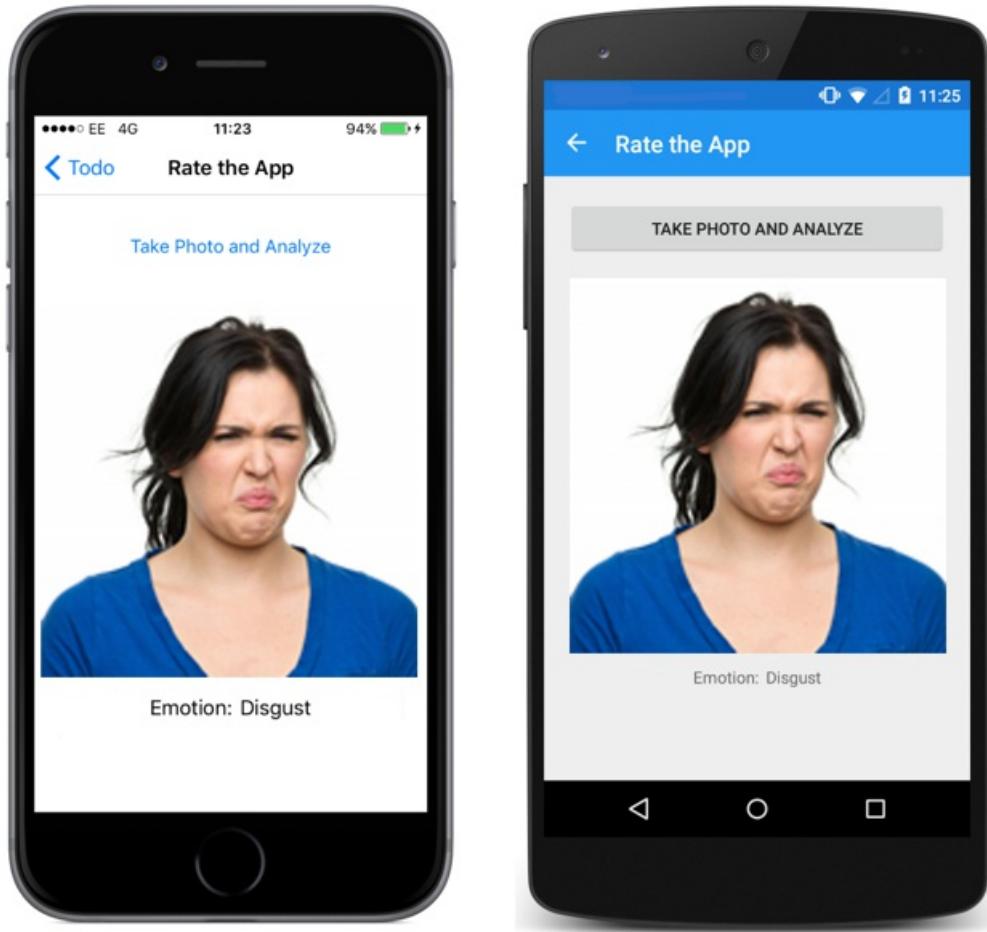
```
[  
  {  
    "faceId": "8a1a80fe-1027-48cf-a7f0-e61c0f005051",  
    "faceRectangle": {  
      "top": 192,  
      "left": 164,  
      "width": 339,  
      "height": 339  
    },  
    "faceAttributes": {  
      "emotion": {  
        "anger": 0.0,  
        "contempt": 0.0,  
        "disgust": 0.0,  
        "fear": 0.0,  
        "happiness": 1.0,  
        "neutral": 0.0,  
        "sadness": 0.0,  
        "surprise": 0.0  
      }  
    }  
  }  
]
```

A successful response message consists of an array of face entries ranked by face rectangle size in descending order, while an empty response indicates no faces detected. Each recognized face includes a series of optional face attributes, which are specified by the `returnFaceAttributes` argument to the `DetectAsync` method.

In the sample application, the JSON response is deserialized into an array of `Face` objects. When interpreting results from the Face API, the detected emotion should be interpreted as the emotion with the highest score, as scores are normalized to sum to one. Therefore, the sample application displays the recognized emotion with the highest score for the largest detected face in the image. This is achieved with the following code:

```
emotionResultLabel.Text = faces.FirstOrDefault().FaceAttributes.Emotion.ToRankedList().FirstOrDefault().Key;
```

The following screenshot shows the result of the emotion recognition process in the sample application:



## Summary

This article explained how to use the Face API to recognize emotion, to rate a Xamarin.Forms application. The Face API takes a facial expression in an image as an input, and returns data that includes the confidence across a set of emotions for each face in the image.

## Related Links

- [Face API](#).
- [Todo Cognitive Services \(sample\)](#)
- [Face REST API](#)

# Xamarin.Forms Deployment and Testing

6/8/2018 • 2 minutes to read • [Edit Online](#)

## Performance

There are many techniques for increasing the performance of Xamarin.Forms apps. Collectively these techniques can greatly reduce the amount of work being performed by a CPU, and the amount of memory consumed by an application.

## Automated Testing with Xamarin.UITest and App Center

Xamarin Test Cloud's **UITest** component can be used with Xamarin.Forms to write UI tests to run in the cloud on hundreds of devices.

# Xamarin.Forms Performance

7/12/2018 • 10 minutes to read • [Edit Online](#)

*There are many techniques for increasing the performance of Xamarin.Forms applications. Collectively these techniques can greatly reduce the amount of work being performed by a CPU, and the amount of memory consumed by an application. This article describes and discusses these techniques.*

## **Evolve 2016: Optimizing App Performance with Xamarin.Forms**

## Overview

Poor application performance presents itself in many ways. It can make an application seem unresponsive, can cause slow scrolling, and can reduce battery life. However, optimizing performance involves more than just implementing efficient code. The user's experience of application performance must also be considered. For example, ensuring that operations execute without blocking the user from performing other activities can help to improve the user's experience.

There are a number of techniques for increasing the performance, and perceived performance, of a Xamarin.Forms application. They include:

- [Enable the XAML Compiler](#)
- [Choose the Correct Layout](#)
- [Enable Layout Compression](#)
- [Use Fast Renderers](#)
- [Reduce Unnecessary Bindings](#)
- [Optimize Layout Performance](#)
- [Optimize ListView Performance](#)
- [Optimize Image Resources](#)
- [Reduce the Visual Tree Size](#)
- [Reduce the Application Resource Dictionary Size](#)
- [Use the Custom Renderer Pattern](#)

### **NOTE**

Before reading this article you should first read [Cross-Platform Performance](#), which discusses non-platform specific techniques to improve the memory usage and performance of applications built using the Xamarin platform.

## Enable the XAML Compiler

XAML can be optionally compiled directly into intermediate language (IL) with the XAML compiler (XAMLC). XAMLC offers a number of benefits:

- It performs compile-time checking of XAML, notifying the user of any errors.
- It removes some of the load and instantiation time for XAML elements.
- It helps to reduce the file size of the final assembly by no longer including .xaml files.

XAMLC is disabled by default to ensure backwards compatibility. However, it can be enabled at both the assembly and class level. For more information, see [Compiling XAML](#).

# Choose the Correct Layout

A layout that's capable of displaying multiple children, but that only has a single child, is wasteful. For example, the following code example shows a `StackLayout` with a single child:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DisplayImage.HomePage">
    <ContentPage.Content>
        <StackLayout>
            <Image Source="waterfront.jpg" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

This is wasteful and the `StackLayout` element should be removed, as shown in the following code example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DisplayImage.HomePage">
    <ContentPage.Content>
        <Image Source="waterfront.jpg" />
    </ContentPage.Content>
</ContentPage>
```

In addition, don't attempt to reproduce the appearance of a specific layout by using combinations of other layouts, as this results in unnecessary layout calculations being performed. For example, don't attempt to reproduce a `Grid` layout by using a combination of `StackLayout` instances. The following code example shows an example of this bad practice:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Details.HomePage"
    Padding="0,20,0,0">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Name:" />
                <Entry Placeholder="Enter your name" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Age:" />
                <Entry Placeholder="Enter your age" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Occupation:" />
                <Entry Placeholder="Enter your occupation" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Address:" />
                <Entry Placeholder="Enter your address" />
            </StackLayout>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

This is wasteful because unnecessary layout calculations are performed. Instead, the desired layout can be better achieved using a `Grid`, as shown in the following code example:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Details.HomePage"
    Padding="0,20,0,0">
    <ContentPage.Content>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
            </Grid.RowDefinitions>
            <Label Text="Name:" />
            <Entry Grid.Column="1" Placeholder="Enter your name" />
            <Label Grid.Row="1" Text="Age:" />
            <Entry Grid.Row="1" Grid.Column="1" Placeholder="Enter your age" />
            <Label Grid.Row="2" Text="Occupation:" />
            <Entry Grid.Row="2" Grid.Column="1" Placeholder="Enter your occupation" />
            <Label Grid.Row="3" Text="Address:" />
            <Entry Grid.Row="3" Grid.Column="1" Placeholder="Enter your address" />
        </Grid>
    </ContentPage.Content>
</ContentPage>

```

## Enable Layout Compression

Layout compression removes specified layouts from the visual tree, in an attempt to improve page rendering performance. The performance benefit that this delivers varies depending on the complexity of a page, the version of the operating system being used, and the device on which the application is running. However, the biggest performance gains will be seen on older devices. For more information, see [Layout Compression](#).

## Use Fast Renderers

Fast renderers reduce the inflation and rendering costs of Xamarin.Forms controls on Android by flattening the resulting native control hierarchy. This further improves performance by creating fewer objects, which in turns results in a less complex visual tree, and less memory use. For more information, see [Fast Renderers](#).

## Reduce Unnecessary Bindings

Don't use bindings for content that can easily be set statically. There is no advantage in binding data that doesn't need to be bound, because bindings aren't cost efficient. For example, setting `Button.Text = "Accept"` has less overhead than binding `Button.Text` to a ViewModel `string` property with value "Accept".

## Optimize Layout Performance

Xamarin.Forms 2 introduced an optimized layout engine that impacts layout updates. To obtain the best possible layout performance, follow these guidelines:

- Reduce the depth of layout hierarchies by specifying `Margin` property values, allowing the creation of layouts with fewer wrapping views. For more information, see [Margins and Padding](#).
- When using a `Grid`, try to ensure that as few rows and columns as possible are set to `Auto` size. Each auto-sized row or column will cause the layout engine to perform additional layout calculations. Instead, use fixed size rows and columns if possible. Alternatively, set rows and columns to occupy a proportional amount of space with the `GridUnitType.Star` enumeration value, provided that the parent tree follows these layout

guidelines.

- Don't set the `VerticalOptions` and `HorizontalOptions` properties of a layout unless required. The default values of `LayoutOptions.Fill` and `LayoutOptions.FillAndExpand` allow for the best layout optimization. Changing these properties has a cost and consumes memory, even when setting them to the default values.
- Avoid using a `RelativeLayout` whenever possible. It will result in the CPU having to perform significantly more work.
- When using an `AbsoluteLayout`, avoid using the `AbsoluteLayout.AutoSize` property whenever possible.
- When using a `StackLayout`, ensure that only one child is set to `LayoutOptions.Expands`. This property ensures that the specified child will occupy the largest space that the `StackLayout` can give to it, and it is wasteful to perform these calculations more than once.
- Don't call any of the methods of the `Layout` class, as they result in expensive layout calculations being performed. Instead, it's likely that the desired layout behavior can be obtained by setting the `TranslationX` and `TranslationY` properties. Alternatively, subclass the `Layout<View>` class to achieve the desired layout behavior.
- Don't update any `Label` instances more frequently than required, as the change of size of the label can result in the entire screen layout being re-calculated.
- Don't set the `Label.VerticalTextAlignment` property unless required.
- Set the `LineBreakMode` of any `Label` instances to `Nowrap` whenever possible.

## Optimize ListView Performance

When using a `ListView` control there are a number of user experiences that should be optimized:

- **Initialization** – the time interval starting when the control is created, and ending when items are shown on screen.
- **Scrolling** – the ability to scroll through the list and ensure that the UI doesn't lag behind touch gestures.
- **Interaction** for adding, deleting, and selecting items.

The `ListView` control requires an application to supply data and cell templates. How this is achieved will have a large impact on the performance of the control. For more information, see [ListView Performance](#).

## Optimize Image Resources

Displaying image resources can greatly increase the app's memory footprint. Therefore, they should only be created when required and should be released as soon as the application no longer requires them. For example, if an application is displaying an image by reading its data from a stream, ensure that stream is created only when it's required, and ensure that the stream is released when it's no longer required. This can be achieved by creating the stream when the page is created, or when the `Page.Appearing` event fires, and then disposing of the stream when the `Page.Disappearing` event fires.

When downloading an image for display with the `ImageSource.FromUri` method, cache the downloaded image by ensuring that the `UriImageSource.CachingEnabled` property is set to `true`. For more information, see [Working with Images](#).

For more information, see [Optimize Image Resources](#).

## Reduce the Visual Tree Size

Reducing the number of elements on a page will make the page render faster. There are two main techniques for achieving this. The first is to hide elements that aren't visible. The `IsVisible` property of each element determines whether the element should be part of the visual tree or not. Therefore, if an element isn't visible because it's hidden behind other elements, either remove the element or set its `IsVisible` property to `false`.

The second technique is to remove unnecessary elements. For example, the following code example shows a page

layout that displays a series of `Label` elements:

```
<ContentPage.Content>
    <StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Hello" />
        </StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Welcome to the App!" />
        </StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Downloading Data..." />
        </StackLayout>
    </StackLayout>
</ContentPage.Content>
```

The same page layout can be maintained with a reduced element count, as shown in the following code example:

```
<ContentPage.Content>
    <StackLayout Padding="20,20,0,0" Spacing="25">
        <Label Text="Hello" />
        <Label Text="Welcome to the App!" />
        <Label Text="Downloading Data..." />
    </StackLayout>
</ContentPage.Content>
```

## Reduce the Application Resource Dictionary Size

Any resources that are used throughout the application should be stored in the application's resource dictionary to avoid duplication. This will help to reduce the amount of XAML that has to be parsed throughout the application.

The following code example shows the `HeadingLabelStyle` resource, which is used application wide, and so is defined in the application's resource dictionary:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Resources.App">
    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="HeadingLabelStyle" TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

However, XAML that's specific to a page shouldn't be included in the app's resource dictionary, as the resources will then be parsed at application startup instead of when required by a page. If a resource is used by a page that's not the startup page, it should be placed in the resource dictionary for that page, therefore helping to reduce the XAML that's parsed when the application starts. The following code example shows the `HeadingLabelStyle` resource, which is only on a single page, and so is defined in the page's resource dictionary:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Test.HomePage"
    Padding="0,20,0,0">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="HeadingLabelStyle" TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        ...
    </ContentPage.Content>
</ContentPage>

```

For more information about application resources, see [Working with Styles](#).

## Use the Custom Renderer Pattern

Most renderer classes expose the `OnElementChanged` method, which is called when a Xamarin.Forms custom control is created to render the corresponding native control. Custom renderer classes, in each platform-specific renderer class, then override this method to instantiate and customize the native control. The `SetNativeControl` method is used to instantiate the native control, and this method will also assign the control reference to the `Control` property.

However, in some circumstances the `OnElementChanged` method can be called multiple times. Therefore, to prevent memory leaks, which can have a performance impact, care must be taken when instantiating a new native control. The approach to use when instantiating a new native control in a custom renderer is shown in the following code example:

```

protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
    base.OnElementChanged (e);

    if (Control == null) {
        // Instantiate the native control
    }

    if (e.OldElement != null) {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null) {
        // Configure the control and subscribe to event handlers
    }
}

```

A new native control should only be instantiated once, when the `Control` property is `null`. The control should only be configured and event handlers subscribed to when the custom renderer is attached to a new Xamarin.Forms element. Similarly, any event handlers that were subscribed to should only be unsubscribed from when the element renderer is attached to changes. Adopting this approach will help to create an efficiently performing custom renderer that doesn't suffer from memory leaks.

For more information about custom renderers, see [Customizing Controls on Each Platform](#).

## Summary

This article described and discussed techniques for increasing the performance of Xamarin.Forms applications. Collectively these techniques can greatly reduce the amount of work being performed by a CPU, and the amount of memory consumed by an application.

## Related Links

- [Cross-Platform Performance](#)
- [ListView Performance](#)
- [Fast Renderers](#)
- [Layout Compression](#)
- [Xamarin.Forms Image Resizer Sample](#)
- [XamlCompilation](#)
- [XamlCompilationOptions](#)

# Automate Xamarin.Forms testing with App Center

11/20/2018 • 5 minutes to read • [Edit Online](#)

Xamarin.UITest component can be used with Xamarin.Forms to write UI tests to run in the cloud on hundreds of devices.

## Overview

**App Center Test** allows developers to write automated user interface tests for iOS and Android apps. With some minor tweaks, Xamarin.Forms apps can be tested using Xamarin.UITest, including sharing the same test code. This article introduces specific tips to get Xamarin.UITest working with Xamarin.Forms.

This guide does assume that familiarity with Xamarin.UITest. The following guides are recommended for gaining familiarity with Xamarin.UITest:

- [Introduction to App Center Test](#)
- [Introduction to UITest](#)

Once a UITest project has been added to a Xamarin.Forms solution, the steps for writing and running the tests for a Xamarin.Forms application are the same as for a Xamarin.Android or Xamarin.iOS application.

## Requirements

Refer to the [Xamarin.UITest](#) to confirm your project is ready for automated UI testing.

## Adding UITest Support to Xamarin.Forms Apps

UITest automates the user interface by activating controls on the screen and performing input anywhere a user would normally interact with the application. To enable tests that can *press a button* or *enter text in a box* the test code will need a way to identify the controls on the screen.

To enable the UITest code to reference controls, each control needs a unique identifier. In Xamarin.Forms, the recommended way to set this identifier is by using the `AutomationId` property as shown below:

```
var b = new Button {
    Text = "Click me",
    AutomationId = "MyButton"
};
var l = new Label {
    Text = "Hello, Xamarin.Forms!",
    AutomationId = "MyLabel"
};
```

The `AutomationId` property can also be set in XAML:

```
<Button x:Name="b" AutomationId="MyButton" Text="Click me"/>
<Label x:Name="l" AutomationId="MyLabel" Text="Hello, Xamarin.Forms!" />
```

### NOTE

`AutomationId` is a `BindableProperty` and so can also be set with a binding expression.

A unique `AutomationId` should be added to all controls that are required for testing (including buttons, text entries, and labels whose value might need to be queried).

[!WARN] Note that an `InvalidOperationException` will be thrown if an attempt is made to set the `AutomationId` property of an `Element` more than once.

## iOS Application Project

To run tests on iOS ,the [Xamarin Test Cloud Agent NuGet package](#) must be added to the project. Once it has been added, copy the following code into the `AppDelegate.FinishedLaunching` method:

```
#if ENABLE_TEST_CLOUD
// requires Xamarin Test Cloud Agent
Xamarin.Calabash.Start();
#endif
```

The Calabash assembly makes uses of non-public Apple API's which will cause apps to be rejected by the App Store. However, the Xamarin.iOS linker will remove the Calabash assembly from the final IPA if it isn't explicitly referenced from code.

### NOTE

Release builds do not have the `ENABLE_TEST_CLOUD` compiler variable, which will cause the Calabash assembly to be removed from app bundle. However, debug builds do have the compiler directive defined, preventing the linker from removing the assembly.

The following screenshot shows the `ENABLE_TEST_CLOUD` compiler variable set for Debug builds:

Configuration: Active (Debug) Platform: Active (iPhone)

General

Conditional compilation symbols: `_UNIFIED_:_MOBILE_:_IOS_:_ENABLE_TEST_CLOUD;`

Define DEBUG constant

Define TRACE constant

Platform target: Any CPU

Prefer 32-bit

Allow unsafe code

Optimize code

Errors and warnings

Warning level: 4

SUPPRESS Warnings:

Treat warnings as errors

None

All

Specific warnings:

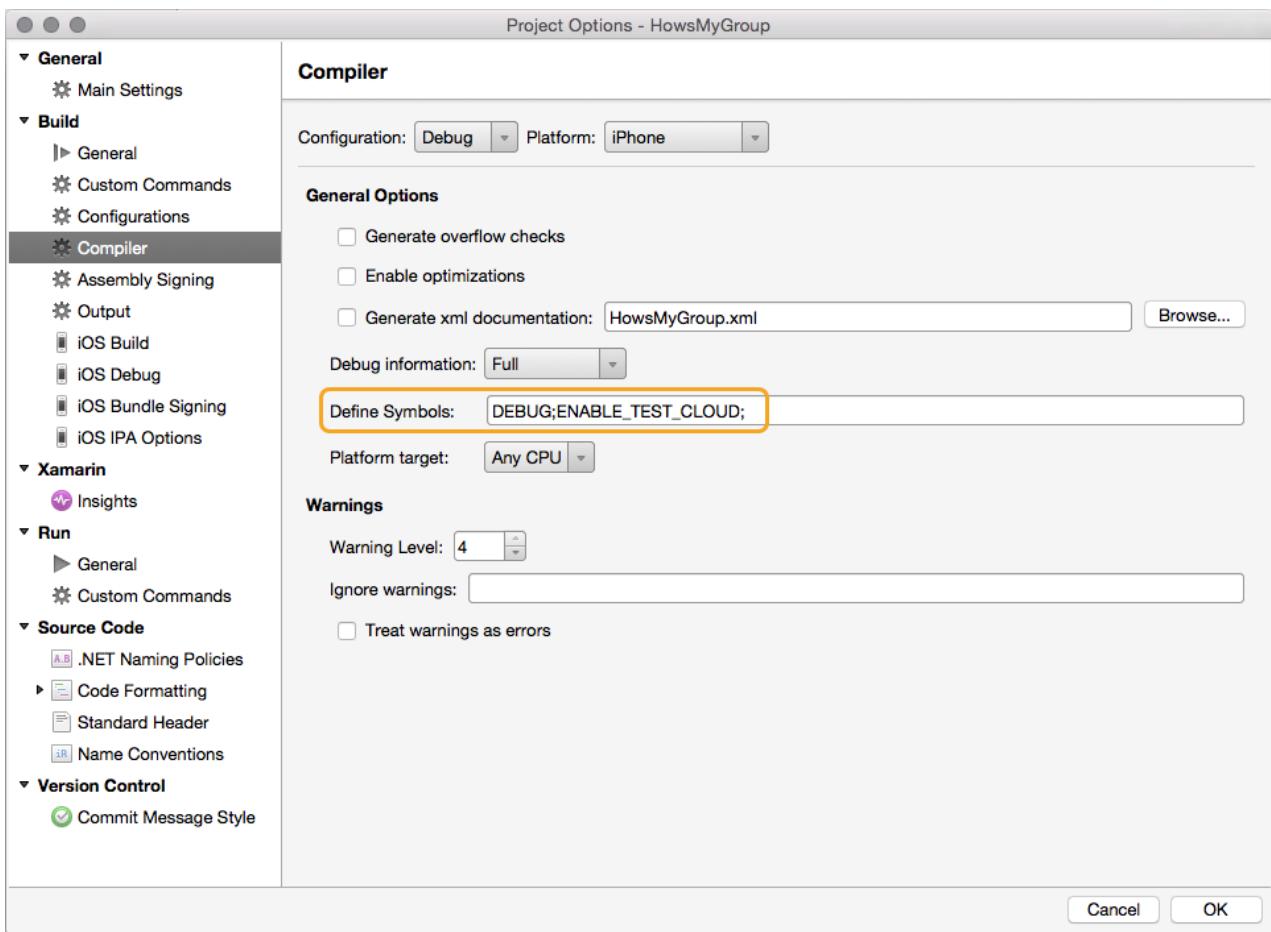
Output

Output path: bin\iPhone\Debug\

XML documentation file:

Register for COM interop

Generate serialization assembly: Auto



## Android Application Project

Unlike iOS, Android projects do not need any special startup code.

## Writing UITests

For information about writing UITests, see [UITest documentation](#). The steps below are a summary, specifically describing how the [Xamarin.Forms demo UsingUITest](#) is built.

### Use AutomationId in the Xamarin.Forms UI

Before any UITests can be written, the Xamarin.Forms application user interface must be scriptable. Ensure that all controls in the user interface have a `AutomationId` so that they can be referenced in test code.

#### Referring to the AutomationId in UITests

When writing UITests, the `AutomationId` value is exposed differently on each platform:

- **iOS** uses the `id` field.
- **Android** uses the `label` field.

To write cross-platform UITests that will find the `AutomationId` on both iOS and Android, use the `Marked` test query:

```
app.Query(c=>c.Marked("MyButton"))
```

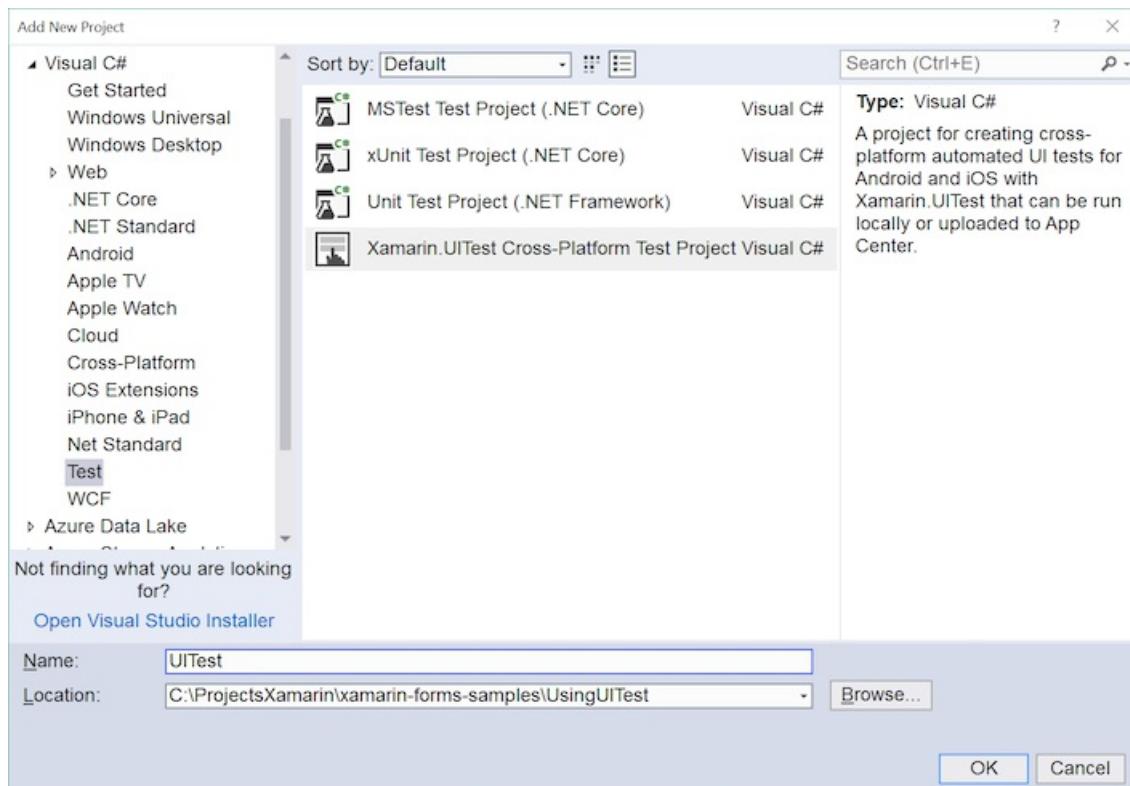
The shorter form `app.Query("MyButton")` also works.

### Adding a UITest Project to an Existing Solution

Visual Studio has a template to help add a Xamarin.UITest project to an existing Xamarin.Forms solution:

1. Right click on the solution, and select **File > New Project**.

2. From the **Visual C#** Templates, select the **Test** category. Select the **UI Test App > Cross-Platform** template:



This will add a new project with the **NUnit**, **Xamarin.UITest**, and **NUnitTestAdapter** NuGet packages to the solution:

NuGet Package Manager: UsingUITest.UITests

Package source: nuget.org Filter: Installed  Include prerelease

Search (Ctrl+E)

 <b>NUnit</b>	NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.	<input checked="" type="checkbox"/>
 <b>NUnitTestAdapter</b>	A package including the NUnit TestAdapter for Visual Studio 2012/13/15. With this package you...	<input checked="" type="checkbox"/>
 <b>Xamarin.UITest</b>	UI Automation Framework for testing Android and iOS apps.	<input checked="" type="checkbox"/>

**Xamarin.UITest**

Action:  Version: 1.1.1

Installed version:

Uninstall

---

**Options**

Show preview window  
 Remove dependencies  
 Force uninstall, even if there are dependencies on it  
[Learn about Options](#)

---

**Description**  
UI Automation Framework for testing Android and iOS apps.

**Author(s):** Xamarin  
**License:** <http://developer.xamarin.com/guides/testcloud/uitest/license/>  
**Project URL:** <http://developer.xamarin.com/guides/testcloud/uitest/>  
**Report Abuse:** <https://www.nuget.org/packages/Xamarin.UITest/1.1.1/ReportAbuse>

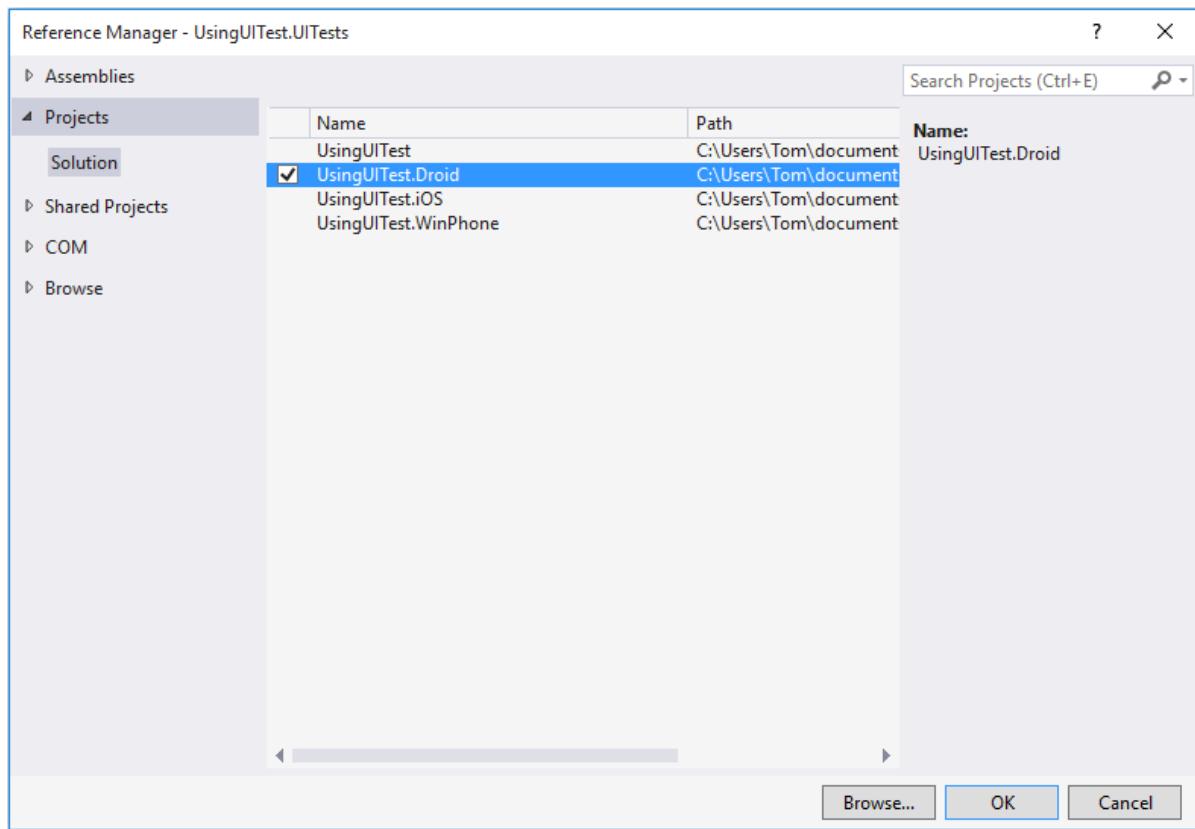
**Tags:**

**Dependencies**  
No dependencies

The **NUnitTestAdapter** is a third party test runner that allows Visual Studio to run NUnit tests from Visual Studio.

The new project also has two classes in it. **AppInitializer** contains code to help initialize and setup tests. The other class, **Tests**, contains boilerplate code to help start the UITests.

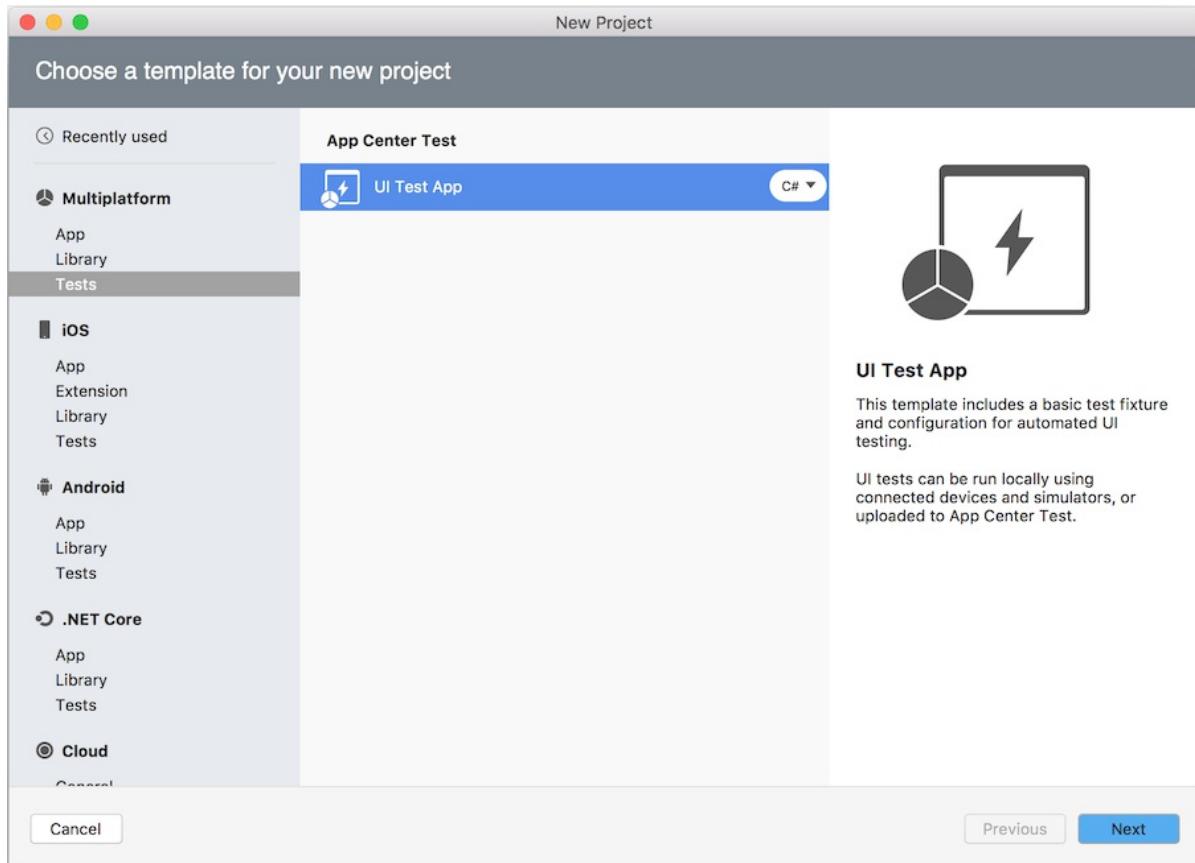
3. Add a project reference from the UITest project to the Xamarin.Android project:



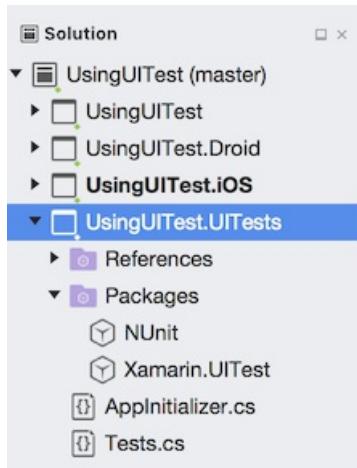
This will allow the **NUnitTestAdapter** to run the UITests for the Android app from Visual Studio.

It is possible to add a new Xamarin.UITest project to an existing solution manually:

1. Start by adding a new project by selecting the solution, and clicking **File > Add New Project**. In the **New Project** dialog, select **Cross-platform > Tests > Xamarin Test Cloud > UI Test App**:

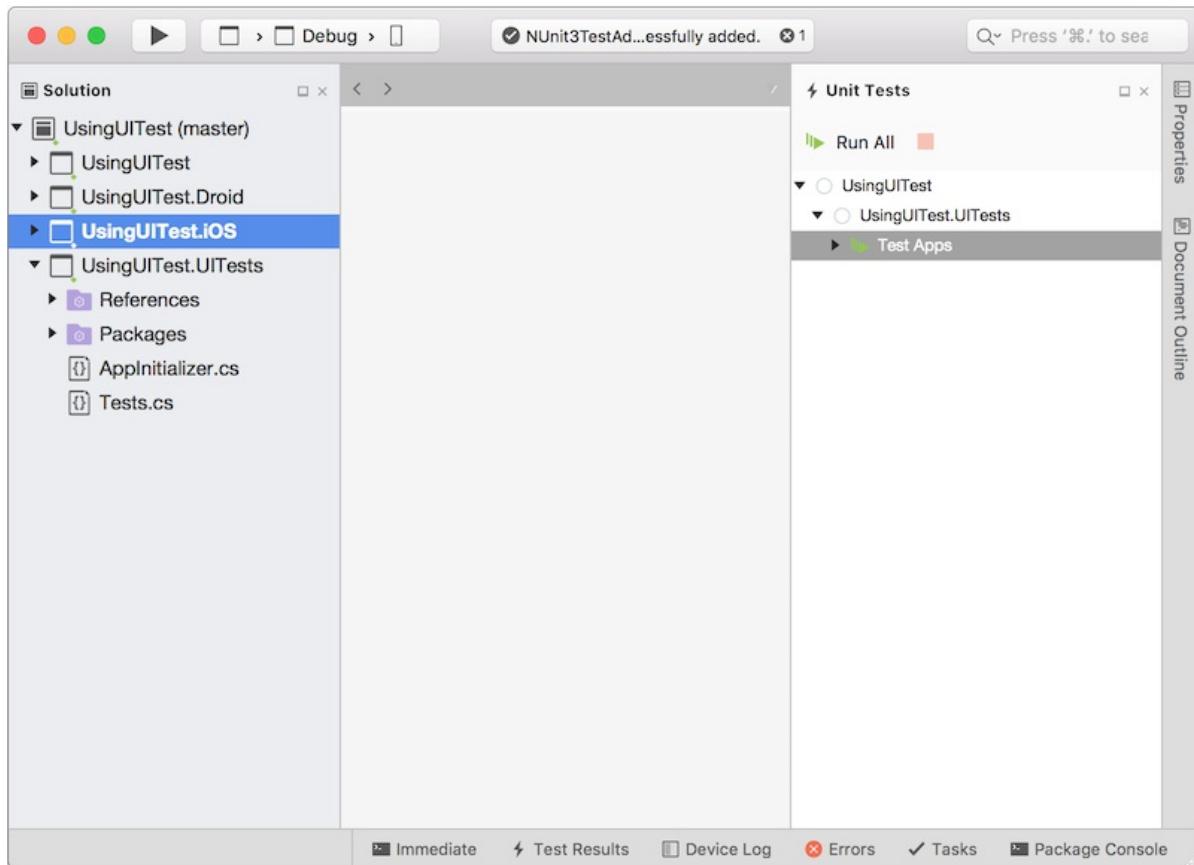


This will add a new project that already has the **NUnit** and **Xamarin.UITest** NuGet packages in the solution:

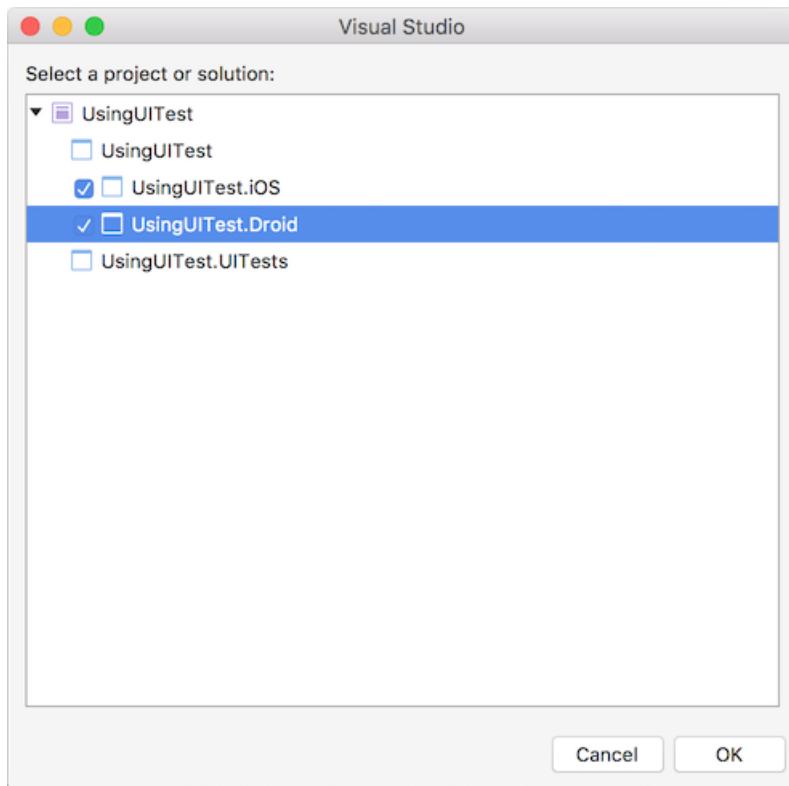


The new project also has two classes in it. **AppInitializer** contains code to help initialize and setup tests. The other class, **Tests**, contains boilerplate code to help start the UITests.

2. Select **View > Pads > Unit Tests** to display the Unit Test pad. Expand **UsingUITest > UsingUITest.UITests > Test Apps**:



3. Right click on **Test Apps**, click on **Add App Project**, and select iOS and Android projects in the dialog that appears:

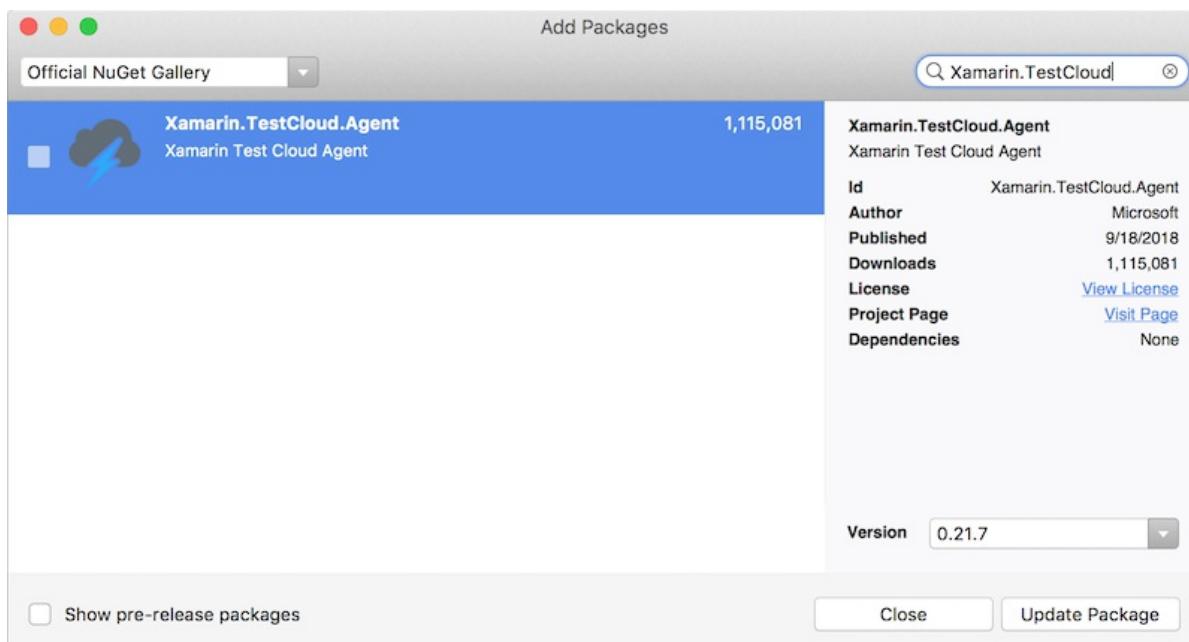


The **Unit Test** pad should now have a reference to the iOS and Android projects. This will allow the Visual Studio for Mac test runner to execute UITests locally against the two Xamarin.Forms projects.

#### Adding UITest to the iOS App

There are some additional changes that need to be performed to the iOS application before Xamarin.UITest will work:

1. Add the **Xamarin Test Cloud Agent** NuGet package. Right click on **Packages**, select **Add Packages**, search NuGet for the **Xamarin Test Cloud Agent** and add it to the Xamarin.iOS project:



2. Edit the `FinishedLaunching` method of the **AppDelegate** class to initialize the Xamarin Test Cloud Agent when the iOS application starts, and to set the `AutomationId` property of the views. The `FinishedLaunching` method should resemble the following code example:

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    #if ENABLE_TEST_CLOUD
    Xamarin.Calabash.Start();
    #endif

    global::Xamarin.Forms.Forms.Init();

    LoadApplication(new App());

    return base.FinishedLaunching(app, options);
}
```

After adding `Xamarin.UITest` to the `Xamarin.Forms` solution, it's possible to create `UITests`, run them locally, and submit them to `Xamarin Test Cloud`.

## Summary

`Xamarin.Forms` applications can be easily tested with **Xamarin.UITest** using a simple mechanism to expose the `AutomationId` as a unique view identifier for test automation. Once a `UITest` project has been added to a `Xamarin.Forms` solution, the steps for writing and running the tests for a `Xamarin.Forms` application are the same as for a `Xamarin.Android` or `Xamarin.iOS` application.

For information about how to submit tests to App Center Test, see [Submitting UITests](#). For more information about `UITest`, see [App Center Test documentation](#).

## Related Links

- [UITestSample](#)
- [Xamarin.Forms Samples](#)
- [App Center Test](#)
- [Xamarin.UITest](#)
- [NUnit](#)

# Advanced Concepts & Internals

7/30/2018 • 2 minutes to read • [Edit Online](#)

## Fast Renderers

This article introduces fast renderers, which reduce the inflation and rendering costs of a Xamarin.Forms control on Android by flattening the resulting native control hierarchy.

## .NET Standard

This article explains how to convert a Xamarin.Forms application to use .NET Standard 2.0.

## Dependency Resolution

This article explains how to inject a dependency resolution method into Xamarin.Forms so that an application's dependency injection container has control over the creation and lifetime of custom renderers, effects, and `DependencyService` implementations.

# Xamarin.Forms Fast Renderers

10/10/2018 • 2 minutes to read • [Edit Online](#)

This article introduces fast renderers, which reduce the inflation and rendering costs of a Xamarin.Forms control on Android by flattening the resulting native control hierarchy.

Traditionally, most of the original control renderers on Android are composed of two views:

- A native control, such as a `Button` or `TextView`.
- A container `ViewGroup` that handles some of the layout work, gesture handling, and other tasks.

However, this approach has a performance implication in that two views are created for each logical control, which results in a more complex visual tree that requires more memory, and more processing to render on screen.

Fast renderers reduce the inflation and rendering costs of a Xamarin.Forms control into a single view. Therefore, instead of creating two views and adding them to the view tree, only one is created. This improves performance by creating fewer objects, which in turn means a less complex view tree, and less memory use (which also results in fewer garbage collection pauses).

Fast renderers are available for the following controls in Xamarin.Forms 2.4 on Android:

- `Button`
- `Image`
- `Label`
- `Frame`

Functionally, these fast renderers are no different to the original renderers. However, they are currently experimental and can only be used by adding the following line of code to your `MainActivity` class before calling `Forms.Init`:

```
Forms.SetFlags("FastRenderers_Experimental");
```

## NOTE

Fast renderers are only applicable to the app compat Android backend, so this setting will be ignored on pre-app compat activities.

Performance improvements will vary for each application, depending upon the complexity of the layout. For example, performance improvements of x2 are possible when scrolling through a `ListView` containing thousands of rows of data, where the cells in each row are made of controls that use fast renderers, which results in visibly smoother scrolling.

## Related Links

- [Custom Renderers](#)

# .NET Standard 2.0 Support in Xamarin.Forms

6/8/2018 • 2 minutes to read • [Edit Online](#)

This article explains how to convert a Xamarin.Forms application to use .NET Standard 2.0.

.NET Standard is a specification of .NET APIs that are intended to be available on all .NET implementations. It makes it easier to share code across desktop applications, mobile apps and games, and cloud services, by bringing identical APIs to the different platforms. For information about the platforms supported by .NET Standard, see [.NET implementation support](#).

.NET Standard libraries are the replacement for Portable Class Libraries (PCL). However, a library that targets .NET Standard is still a PCL, and is referred to as a .NET Standard-based PCL. Certain PCL profiles are mapped to .NET Standard versions, and for profiles that have a mapping, the two library types will be able to reference each other. For more information, see [PCL compatibility](#).

Xamarin.Forms 2.4 allows Xamarin.Forms applications to target .NET Standard 2.0 by replacing the PCL with a .NET Standard 2.0 library. This can be achieved as follows:

- Ensure [.NET Core 2.0](#) is installed.
- Update the Xamarin.Forms solution to use Xamarin.Forms 2.4, or greater.
- Add a .NET Standard library to the solution, that targets .NET Standard 2.0.
- Delete the class that's added to the .NET Standard library.
- Add the Xamarin.Forms 2.4 (or greater) NuGet package to the .NET Standard library.
- In the platform projects, add a reference to the .NET Standard library and remove the reference to the PCL project that contains the Xamarin.Forms user interface logic.
- Copy the files from the PCL project to the .NET Standard library.
- Remove the PCL project that contains the Xamarin.Forms user interface logic.

## Related Links

- [.NET Standard](#)
- [Code Sharing Options](#)

# Dependency resolution in Xamarin.Forms

11/20/2018 • 9 minutes to read • [Edit Online](#)

This article explains how to inject a dependency resolution method into Xamarin.Forms so that an application's dependency injection container has control over the creation and lifetime of custom renderers, effects, and `DependencyService` implementations. The code examples in this article are taken from the [Dependency Resolution using Containers](#) sample.

In the context of a Xamarin.Forms application that uses the Model-View-ViewModel (MVVM) pattern, a dependency injection container can be used for registering and resolving view models, and for registering services and injecting them into view models. During view model creation, the container injects any dependencies that are required. If those dependencies have not been created, the container creates and resolves the dependencies first. For more information about dependency injection, including examples of injecting dependencies into view models, see [Dependency Injection](#).

Control over the creation and lifetime of types in platform projects is traditionally performed by Xamarin.Forms, which uses the `Activator.CreateInstance` method to create instances of custom renderers, effects, and `DependencyService` implementations. Unfortunately, this limits developer control over the creation and lifetime of these types, and the ability to inject dependencies into them. This behavior can be changed by injecting a dependency resolution method into Xamarin.Forms that controls how types will be created – either by the application's dependency injection container, or by Xamarin.Forms. However, note that there is no requirement to inject a dependency resolution method into Xamarin.Forms. Xamarin.Forms will continue to create and manage the lifetime of types in platform projects if a dependency resolution method isn't injected.

## NOTE

While this article focuses on injecting a dependency resolution method into Xamarin.Forms that resolves registered types using a dependency injection container, it's also possible to inject a dependency resolution method that uses factory methods to resolve registered types. For more information, see the [Dependency Resolution using Factory Methods](#) sample.

## Injecting a dependency resolution method

The `DependencyResolver` class provides the ability to inject a dependency resolution method into Xamarin.Forms, using the `ResolveUsing` method. Then, when Xamarin.Forms needs an instance of a particular type, the dependency resolution method is given the opportunity to provide the instance. If the dependency resolution method returns `null` for a requested type, Xamarin.Forms falls back to attempting to create the type instance itself using the `Activator.CreateInstance` method.

The following example shows how to set the dependency resolution method with the `ResolveUsing` method:

```

using Autofac;
using Xamarin.Forms.Internals;
...

public partial class App : Application
{
    //.IContainer and ContainerBuilder are provided by Autofac
    static IContainer container;
    static readonly ContainerBuilder builder = new ContainerBuilder();

    public App()
    {
        ...
        DependencyResolver.ResolveUsing(type => container.IsRegistered(type) ? container.Resolve(type) :
null);
        ...
    }
    ...
}

```

In this example, the dependency resolution method is set to a lambda expression that uses the Autofac dependency injection container to resolve any types that have been registered with the container. Otherwise, `null` will be returned, which will result in Xamarin.Forms attempting to resolve the type.

#### **NOTE**

The API used by a dependency injection container is specific to the container. The code examples in this article use Autofac as a dependency injection container, which provides the `IContainer` and `ContainerBuilder` types. Alternative dependency injection containers could equally be used, but would use different APIs than are presented here.

Note that there is no requirement to set the dependency resolution method during application startup. It can be set at any time. The only constraint is that Xamarin.Forms needs to know about the dependency resolution method by the time that the application attempts to consume types stored in the dependency injection container. Therefore, if there are services in the dependency injection container that the application will require during startup, the dependency resolution method will have to be set early in the application's lifecycle. Similarly, if the dependency injection container manages the creation and lifetime of a particular `Effect`, Xamarin.Forms will need to know about the dependency resolution method before it attempts to create a view that uses that `Effect`.

#### **WARNING**

Registering and resolving types with a dependency injection container has a performance cost because of the container's use of reflection for creating each type, especially if dependencies are being reconstructed for each page navigation in the application. If there are many or deep dependencies, the cost of creation can increase significantly.

## Registering types

Types must be registered with the dependency injection container before it can resolve them via the dependency resolution method. The following code example shows the registration methods that the sample application exposes in the `App` class, for the Autofac container:

```

using Autofac;
using Autofac.Core;
...

public partial class App : Application
{
    static IContainer container;
    static readonly ContainerBuilder builder = new ContainerBuilder();
    ...

    public static void RegisterType<T>() where T : class
    {
        builder.RegisterType<T>();
    }

    public static void RegisterType<TInterface, T>() where TInterface : class where T : class, TInterface
    {
        builder.RegisterType<T>().As<TInterface>();
    }

    public static void RegisterTypeWithParameters<T>(Type param1Type, object param1Value, Type param2Type,
string param2Name) where T : class
    {
        builder.RegisterType<T>()
            .WithParameters(new List<Parameter>())
        {
            new TypedParameter(param1Type, param1Value),
            new ResolvedParameter(
                (pi, ctx) => pi.ParameterType == param2Type && pi.Name == param2Name,
                (pi, ctx) => ctx.Resolve(param2Type))
        });
    }

    public static void RegisterTypeWithParameters<TInterface, T>(Type param1Type, object param1Value, Type param2Type,
string param2Name) where TInterface : class where T : class, TInterface
    {
        builder.RegisterType<T>()
            .WithParameters(new List<Parameter>())
        {
            new TypedParameter(param1Type, param1Value),
            new ResolvedParameter(
                (pi, ctx) => pi.ParameterType == param2Type && pi.Name == param2Name,
                (pi, ctx) => ctx.Resolve(param2Type))
        }).As<TInterface>();
    }

    public static void BuildContainer()
    {
        container = builder.Build();
    }
    ...
}

```

When an application uses a dependency resolution method to resolve types from a container, type registrations are typically performed from platform projects. This enables platform projects to register types for custom renderers, effects, and `DependencyService` implementations.

Following type registration from a platform project, the `IContainer` object must be built, which is accomplished by calling the `BuildContainer` method. This method invokes Autofac's `Build` method on the `ContainerBuilder` instance, which builds a new dependency injection container that contains the registrations that have been made.

In the sections that follow, a `Logger` class that implements the `ILogger` interface is injected into class constructors. The `Logger` class implements simple logging functionality using the `Debug.WriteLine` method, and is used to demonstrate how services can be injected into custom renderers, effects, and `DependencyService` implementations.

## Registering custom renderers

The sample application includes a page that plays web videos, whose XAML source is shown in the following example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:video="clr-namespace:FormsVideoLibrary"
    ...
<video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />
</ContentPage>
```

The `VideoPlayer` view is implemented on each platform by a `VideoPlayerRenderer` class, that provides the functionality for playing the video. For more information about these custom renderer classes, see [Implementing a video player](#).

On iOS and the Universal Windows Platform (UWP), the `VideoPlayerRenderer` classes have the following constructor, which requires an `ILogger` argument:

```
public VideoPlayerRenderer(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

On all the platforms, type registration with the dependency injection container is performed by the `RegisterTypes` method, which is invoked prior to the platform loading the application with the `LoadApplication(new App())` method. The following example shows the `RegisterTypes` method on the iOS platform:

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterType<FormsVideoLibrary.iOS.VideoPlayerRenderer>();
    App.BuildContainer();
}
```

In this example, the `Logger` concrete type is registered via a mapping against its interface type, and the `VideoPlayerRenderer` type is registered directly without an interface mapping. When the user navigates to the page containing the `VideoPlayer` view, the dependency resolution method will be invoked to resolve the `VideoPlayerRenderer` type from the dependency injection container, which will also resolve and inject the `Logger` type into the `VideoPlayerRenderer` constructor.

The `VideoPlayerRenderer` constructor on the Android platform is slightly more complicated as it requires a `Context` argument in addition to the `ILogger` argument:

```
public VideoPlayerRenderer(Context context, ILogger logger) : base(context)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

The following example shows the `RegisterTypes` method on the Android platform:

```

void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterTypeWithParameters<FormsVideoLibrary.Droid.VideoPlayerRenderer>
    (typeof(Android.Content.Context), this, typeof(ILogger), "logger");
    App.BuildContainer();
}

```

In this example, the `App.RegisterTypeWithParameters` method registers the `VideoPlayerRenderer` with the dependency injection container. The registration method ensures that the `MainActivity` instance will be injected as the `Context` argument, and that the `Logger` type will be injected as the `ILogger` argument.

## Registering effects

The sample application includes a page that uses a touch tracking effect to drag `BoxView` instances around the page. The `Effect` is added to the `BoxView` using the following code:

```

var boxView = new BoxView { ... };
var touchEffect = new TouchEffect();
boxView.Effects.Add(touchEffect);

```

The `TouchEffect` class is a `RoutingEffect` that's implemented on each platform by a `TouchEffect` class that's a `PlatformEffect`. The platform `TouchEffect` class provides the functionality for dragging the `BoxView` around the page. For more information about these effect classes, see [Invoking events from effects](#).

On all the platforms, the `TouchEffect` class has the following constructor, which requires an `ILogger` argument:

```

public TouchEffect(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}

```

On all the platforms, type registration with the dependency injection container is performed by the `RegisterTypes` method, which is invoked prior to the platform loading the application with the `LoadApplication(new App())` method. The following example shows the `RegisterTypes` method on the Android platform:

```

void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterType<TouchTracking.Droid.TouchEffect>();
    App.BuildContainer();
}

```

In this example, the `Logger` concrete type is registered via a mapping against its interface type, and the `TouchEffect` type is registered directly without an interface mapping. When the user navigates to the page containing a `BoxView` instance that has the `TouchEffect` attached to it, the dependency resolution method will be invoked to resolve the platform `TouchEffect` type from the dependency injection container, which will also resolve and inject the `Logger` type into the `TouchEffect` constructor.

## Registering DependencyService implementations

The sample application includes a page that uses `DependencyService` implementations on each platform to allow the user to pick a photo from the device's picture library. The `IPhotoPicker` interface defines the functionality that is implemented by the `DependencyService` implementations, and is shown in the following example:

```
public interface IPhotoPicker
{
    Task<Stream> GetImageStreamAsync();
}
```

In each platform project, the `PhotoPicker` class implements the `IPhotoPicker` interface using platform APIs. For more information about these dependency services, see [Picking a photo from the picture library](#).

On iOS and UWP, the `PhotoPicker` classes have the following constructor, which requires an `ILogger` argument:

```
public PhotoPicker(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

On all the platforms, type registration with the dependency injection container is performed by the `RegisterTypes` method, which is invoked prior to the platform loading the application with the `LoadApplication(new App())` method. The following example shows the `RegisterTypes` method on UWP:

```
void RegisterTypes()
{
    DIContainerDemo.App.RegisterType<ILogger, Logger>();
    DIContainerDemo.App.RegisterType<IPhotoPicker, Services.UWP.PhotoPicker>();
    DIContainerDemo.App.BuildContainer();
}
```

In this example, the `Logger` concrete type is registered via a mapping against its interface type, and the `PhotoPicker` type is also registered via a interface mapping.

The `PhotoPicker` constructor on the Android platform is slightly more complicated as it requires a `Context` argument in addition to the `ILogger` argument:

```
public PhotoPicker(Context context, ILogger logger)
{
    _context = context ?? throw new ArgumentNullException(nameof(context));
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

The following example shows the `RegisterTypes` method on the Android platform:

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterTypeWithParameters<IPhotoPicker, Services.Droid.PhotoPicker>(typeof(Android.Content.Context),
    this, typeof(ILogger), "logger");
    App.BuildContainer();
}
```

In this example, the `App.RegisterTypeWithParameters` method registers the `PhotoPicker` with the dependency injection container. The registration method ensures that the `MainActivity` instance will be injected as the `Context` argument, and that the `Logger` type will be injected as the `ILogger` argument.

When the user navigates to the photo picking page and chooses to select a photo, the `onSelectPhotoButtonClicked` handler is executed:

```
async void OnSelectPhotoButtonClicked(object sender, EventArgs e)
{
    ...
    var photoPickerService = DependencyService.Resolve<IPhotoPicker>();
    var stream = await photoPickerService.GetImageStreamAsync();
    if (stream != null)
    {
        image.Source = ImageSource.FromStream(() => stream);
    }
    ...
}
```

When the `DependencyService.Resolve<T>` method is invoked, the dependency resolution method will be invoked to resolve the `PhotoPicker` type from the dependency injection container, which will also resolve and inject the `Logger` type into the `PhotoPicker` constructor.

#### NOTE

The `Resolve<T>` method must be used when resolving a type from the application's dependency injection container via the `DependencyService`.

## Related links

- [Dependency resolution using containers \(sample\)](#)
- [Dependency injection](#)
- [Implementing a video player](#)
- [Invoking events from effects](#)
- [Picking a photo from the picture library](#)

# Troubleshooting

4/12/2018 • 2 minutes to read • [Edit Online](#)

*Common error conditions and how to resolve them*

## Error: "Unable to find a version of Xamarin.Forms compatible with..."

The following errors can appear in the **Package Console** window when updating all the Nuget packages in a Xamarin.Forms solution or in a Xamarin.Forms Android app project:

```
Attempting to resolve dependency 'Xamarin.Android.Support.v7.AppCompat (= 23.3.0.0)'.
Attempting to resolve dependency 'Xamarin.Android.Support.v4 (= 23.3.0.0)'.
Looking for updates for 'Xamarin.Android.Support.v7.MediaRouter'...
Updating 'Xamarin.Android.Support.v7.MediaRouter' from version '23.3.0.0' to '23.3.1.0' in project
'Todo.Droid'.
Updating 'Xamarin.Android.Support.v7.MediaRouter 23.3.0.0' to 'Xamarin.Android.Support.v7.MediaRouter 23.3.1.0'
failed.
Unable to find a version of 'Xamarin.Forms' that is compatible with 'Xamarin.Android.Support.v7.MediaRouter
23.3.0.0'.
```

### What causes this error?

Visual Studio for Mac (or Visual Studio) may indicate that updates are available for the Xamarin.Forms Nuget package *and all its dependencies*. In Xamarin Studio, the solution's **Packages** node might look like this (the version numbers might be different):



This error may occur if you attempt to update *all* the packages.

This is because with Android projects set to a target/compile version of Android 6.0 (API 23) or below, Xamarin.Forms has a hard dependency on *specific* versions of the Android support packages. Although updated versions of those packages may be available, Xamarin.Forms is not necessarily compatible with them.

In this case you should update *only* the **Xamarin.Forms** package as this will ensure that the dependencies remain on compatible versions. Other packages that you have added to your project may also be updated individually as long as they do not cause the Android support packages to update.

**NOTE**

If you are using Xamarin.Forms 2.3.4 or higher **and** your Android project's target/compile version is set to Android 7.0 (API 24) or higher, then the hard dependencies mentioned above no longer apply and you may update the support packages independently of the Xamarin.Forms package.

**Fix: Remove all packages, and re-add Xamarin.Forms**

If the **Xamarin.Android.Support** packages have been updated to incompatible versions, the simplest fix is to:

1. Manually delete all the Nuget packages in the Android project, then
2. Re-add the **Xamarin.Forms** package.

This will automatically download the *correct* versions of the other packages.

To see a video of this process, refer to this [forums post](#).

# Frequently Asked Questions

5/10/2018 • 2 minutes to read • [Edit Online](#)

## Can I update the Xamarin.Forms default template to a newer NuGet package?

This guide uses the Xamarin.Forms .NET Standard library template as an example, but the same general method will also work for the Xamarin.Forms Shared Project template.

## Why doesn't the Visual Studio XAML designer work for Xamarin.Forms XAML files?

Xamarin.Forms doesn't currently support visual designers for XAML files.

## Android build error: The "LinkAssemblies" task failed unexpectedly

You may see an error message `The "LinkAssemblies" task failed unexpectedly` when building a Xamarin.Android project that uses Forms. This happens when the linker is active (typically on a *Release* build to reduce the size of the app package); and it occurs because the Android targets aren't updated to the latest framework.

## "Why does my Xamarin.Forms.Maps Android project fail with COMPILETODALVIK : UNEXPECTED TOP-LEVEL ERROR?"

This error may be seen in the Error pad of Visual Studio for Mac or in the Build Output window of Visual Studio; in Android projects using Xamarin.Forms.Maps. It is most commonly resolved by increasing the Java Heap Size for your Xamarin.Android project.

# Can I update the Xamarin.Forms default template to a newer NuGet package?

5/10/2018 • 2 minutes to read • [Edit Online](#)

This guide uses the Xamarin.Forms .NET Standard library template as an example, but the same general method will also work for the Xamarin.Forms Shared Project template. This guide is written with the example of updating from Xamarin.Forms 1.5.1.6471 to 2.1.0.6529, but the same steps are possible to set other versions as the default instead.

1. Copy the original template `.zip` from:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Xamarin\Xamarin\[Xamarin Version]\T\PT\Cross-Platform\Xamarin.Forms.PCL.zip
```

2. Unzip the `.zip` to a temporary location.
3. Change all of the occurrences of the old version of the Forms package to the new version you'd like to use.

- `FormsTemplate\FormsTemplate.vstemplate`
- `FormsTemplate.Android\FormsTemplate.Android.vstemplate`
- `FormsTemplate.iOS\FormsTemplate.iOS.vstemplate`

Example: `<package id="Xamarin.Forms" version="1.5.1.6471" />` ->  
`<package id="Xamarin.Forms" version="2.1.0.6529" />`

4. Change the "name" element of the main [multi-project template file](#) (`Xamarin.Forms.PCL.vstemplate`) to make it unique. For example:

```
Blank App (Xamarin.Forms Portable) - 2.1.0.6529
```

5. Re-zip the whole template folder. Make sure to match the original file structure of the `.zip` file. The `Xamarin.Forms.PCL.vstemplate` file should be at the top of the `.zip` file, not within any folders.

6. Create a "Mobile Apps" subdirectory in your per-user Visual Studio templates folder:

```
%USERPROFILE%\Documents\Visual Studio 2013\Templates\ProjectTemplates\Visual C#\Mobile Apps
```

7. Copy the new zipped-up template folder into the new "Mobile Apps" directory.

8. Download the NuGet package that matches the version from step 3. For example,  
<http://nuget.org/api/v2/package/Xamarin.Forms/2.1.0.6529> (see also  
<http://stackoverflow.com/questions/8597375/how-to-get-the-url-of-a-nupkg-file>), and copy it into the appropriate subfolder of the Xamarin Visual Studio extensions folder:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Xamarin\Xamarin\[Xamarin Version]\Packages
```

# Why doesn't the Visual Studio XAML designer work for Xamarin.Forms XAML files?

5/24/2018 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms doesn't currently support visual designers for XAML files. Because of this, when trying to open a Forms XAML file in either Visual Studio's *XAML UI Designer* or *XAML UI Designer with Encoding*, the following error message is thrown:

"The file cannot be opened with the selected editor. Please choose another editor."

This limitation is described in the [Overview](#) section of the [Xamarin.Forms XAML Basics](#) guide:

"There is not yet a visual designer for generating XAML in Xamarin.Forms applications, so all XAML must be hand-written."

However, the Xamarin.Forms XAML Previewer can be displayed by selecting the **View > Other Windows > Xamarin.Forms Previewer** menu option.

# Android build error – The LinkAssemblies task failed unexpectedly

10/16/2018 • 2 minutes to read • [Edit Online](#)

You may see an error message `The "LinkAssemblies" task failed unexpectedly` when building a Xamarin.Android project that uses Forms. This happens when the linker is active (typically on a *Release* build to reduce the size of the app package); and it occurs because the Android targets aren't updated to the latest framework. (More information: [Xamarin.Forms for Android Requirements](#))

The resolution to this issue is to make sure you have the latest supported Android SDK versions, and set the **Target Framework to Use latest installed platform**. It's also recommended that you set the **Target Android Version to Use Target Framework Version** and the **minimum Android version** to API 15 or higher. This is considered the supported configuration.

## Setting in Visual Studio for Mac

1. Right click on the Android project.
2. Go to **Build > General > Target Framework**.
3. Set the **Target Framework: Use latest installed platform**.
4. Still in the project options, go to **Build > Android Application**.
5. Set the **Minimum Android version** to API level 15 or higher & the **Target Android version** to **Automatic - use target framework version**.

## Setting in Visual Studio

1. Right click on the Android project.
2. Go to **Application** in the project options.
3. Set the **Compile using Android version** & **Target Android version** settings to **Use Latest Platform / Use Compile using SDK version**.
4. Set the **Minimum Android to target** setting to API 19 or higher.

Once you've updated those settings, please clean and rebuild your project to ensure your changes are picked up.

# Why does my Xamarin.Forms.Maps Android project fail with COMPILETODALVIK UNEXPECTED TOP-LEVEL ERROR?

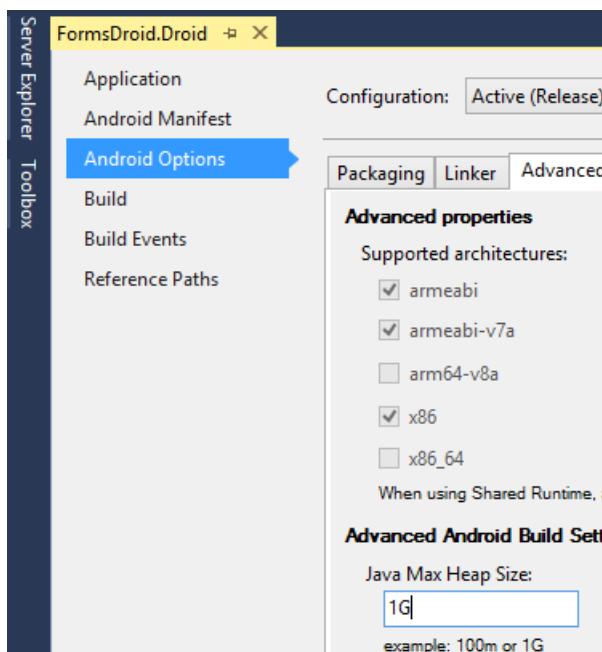
4/12/2018 • 2 minutes to read • [Edit Online](#)

This error may be seen in the Error pad of Visual Studio for Mac or in the Build Output window of Visual Studio; in Android projects using Xamarin.Forms.Maps.

This is most commonly resolved by increasing the Java Heap Size for your Xamarin.Android project. Follow these steps to increase the heap size:

## Visual Studio

1. Right-click the Android project & open the project options.
2. Go to **Android Options -> Advanced**
3. In the Java heap size text box enter 1G.
4. Rebuild the project.



## Visual Studio for Mac

1. Right-click the Android project & open the project options.
2. Go to **Build -> Android Build -> Advanced**
3. In the Java heap size text box enter 1G.
4. Rebuild the project.

General

Main Settings

Build

General

Custom Commands

Configurations

Compiler

Assembly Signing

Output

Android Build

Android Application

Run

General

Custom Commands

Source Code

.NET Naming Policies

Code Formatting

Standard Header

Name Conventions

Version Control

### Android Build

Configuration: Debug Platform: Any CPU

Packaging Linker Advanced

**Supported ABIs**

List of ABIs to support. If no ABI is specified, 'armeabi-v7a' is used.

armeabi  
 armeabi-v7a  
 x86

**Additional Options**

Java heap size: **1G**

Java arguments:

Mandroid arguments:

Cancel OK

# Creating Mobile Apps with Xamarin.Forms book

11/12/2018 • 5 minutes to read • [Edit Online](#)



The book *Creating Mobile Apps with Xamarin.Forms* by Charles Petzold is a guide for learning how to write Xamarin.Forms applications. The only prerequisite is knowledge of the C# programming language. The book provides an extensive exploration into the Xamarin.Forms user interface and also covers animation, MVVM, triggers, behaviors, custom layouts, custom renderers, and much more.

The book was published in the spring of 2016, and has not been updated since then. There is much in the book that remains valuable, but some of the [material is outdated](#), and some topics are no longer entirely correct or complete.

## Download eBook for free

Download your preferred eBook format from Microsoft Virtual Academy:

- [PDF \(56Mb\)](#)
- [ePub \(151Mb\)](#)
- [Kindle edition \(325Mb\)](#)

You can also [download individual chapters](#) as PDF files.

## Samples

The samples are [available on github](#), and include projects for iOS, Android, and the Universal Windows Platform (UWP). (Xamarin.Forms no longer supports Windows 10 Mobile, but Xamarin.Forms applications will run on the Windows 10 desktop.)

## Chapter summaries

Chapter summaries are available in the [chapter table](#) show below. These summaries describe the contents of each chapter, and include several types of links:

- Links to the actual chapters of the book (at the bottom of the page), and to related articles
- Links to all the samples in the [xamarin-forms-book-samples](#) GitHub repository
- Links to the API documentation for more detailed descriptions of Xamarin.Forms classes, structures, properties, enumerations, and so forth

These summaries also indicate when material in the chapter might be [somewhat outdated](#).

## Download chapters and summaries

CHAPTER	COMPLETE TEXT	SUMMARY
Chapter 1. How Does Xamarin.Forms Fit In?	<a href="#">Download PDF</a>	<a href="#">Summary</a>

CHAPTER	COMPLETE TEXT	SUMMARY
Chapter 2. Anatomy of an App	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 3. Deeper into Text	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 4. Scrolling the Stack	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 5. Dealing with Sizes	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 6. Button Clicks	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 7. XAML vs. Code	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 8. Code and XAML in Harmony	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 9. Platform-Specific API Calls	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 10. XAML Markup Extensions	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 11. The Bindable Infrastructure	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 12. Styles	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 13. Bitmaps	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 14. Absolute Layout	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 15. The Interactive Interface	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 16. Data Binding	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 17. Mastering the Grid	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 18. MVVM	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 19. Collection Views	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 20. Async and File I/O	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 21. Transforms	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 22. Animation	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 23. Triggers and Behaviors	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 24. Page Navigation	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 25. Page Varieties	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 26. Custom Layouts	<a href="#">Download PDF</a>	<a href="#">Summary</a>

CHAPTER	COMPLETE TEXT	SUMMARY
Chapter 27. Custom renderers	<a href="#">Download PDF</a>	<a href="#">Summary</a>
Chapter 28. Location and Maps	<a href="#">Download PDF</a>	<a href="#">Summary</a>

## Ways in which the book is outdated

Since the publication of *Creating Mobile Apps with Xamarin.Forms*, several new features have been added to Xamarin.Forms. These new features are described in individual articles in the [Xamarin.Forms](#) documentation.

Other changes have caused some of the content of the book to be outdated:

### .NET Standard 2.0 libraries have replaced Portable Class Libraries

A Xamarin.Forms application generally uses a library to share code among the different platforms. Originally, this was a Portable Class Library (PCL). There are many references to PCLs throughout the book and the chapter summaries.

The Portable Class Library has been replaced with a .NET Standard 2.0 library, as described in the article [.NET Standard 2.0 Support in Xamarin.Forms](#). All the [sample code](#) from the book has been updated to use .NET Standard 2.0 libraries.

Most of the information in the book concerning the role of the Portable Class Library remains the same for a .NET Standard 2.0 library. One difference is that only a PCL has a numeric "profile." Also, there are some advantages to .NET Standard 2.0 libraries. For example, Chapter 20, [Async and File I/O](#) describes how to use the underlying platforms for performing file I/O. This is no longer necessary. The .NET Standard 2.0 library supports the familiar [System.IO](#) classes for all Xamarin.Forms platforms.

The .NET Standard 2.0 library also allows Xamarin.Forms applications to use [HttpClient](#) to access files over the Internet rather than [WebRequest](#) or other classes.

### The role of XAML has been elevated

*Creating Mobile Apps with Xamarin.Forms* begins by describing how to write Xamarin.Forms applications using C#. The Extensible Application Markup Language (XAML) isn't introduced until [Chapter 7. XAML vs. Code](#).

XAML now has a much larger role in Xamarin.Forms. The Xamarin.Forms solution templates distributed with Visual Studio create XAML-based page files. A developer using Xamarin.Forms should become familiar with XAML as early as possible. The [eXtensible Application Markup Language \(XAML\)](#) section of the Xamarin.Forms documentation contains several articles about XAML to get you started.

### Supported platforms

Xamarin.Forms no longer supports Windows 8.1 and Windows Phone 8.1.

The book sometimes makes references to the *Windows Runtime*. This is a term that encompasses the Windows API used in several versions of Windows and Windows Phone. More recent versions of Xamarin.Forms restricts itself to supporting the Universal Windows Platform, which is the API for Windows 10 and Windows 10 Mobile.

A .NET Standard 2.0 library does not support any version of Windows 10 Mobile. Therefore, a Xamarin.Forms application using a .NET Standard library will not run on a Windows 10 Mobile device. Xamarin.Forms applications continue to run on the Windows 10 desktop, versions 10.0.16299.0 and above.

Xamarin.Forms has preview support for the [Mac](#), [WPF](#), [GTK#](#), and [Tizen](#) platforms.

### Chapter summaries

The chapter summaries include information concerning changes in Xamarin.Forms since the book was written. These are often in the form of notes:

#### NOTE

Notes on each page indicate where Xamarin.Forms has diverged from the material presented in the book.

## Samples

In the [xamarin-forms-book-samples](#) GitHub repository, the **original-code-from-book** branch contains program samples consistent with the book. The **master** branch contains projects that have been upgraded to remove deprecated APIs and reflect enhanced APIs. In addition, the Android projects in the **master** branch have been upgraded for Android [Material Design via AppCompat](#) and will generally display black text on a white background.

## Related Links

- [MS Press blog](#)
- [Sample code from book](#)

# Enterprise Application Patterns using Xamarin.Forms eBook

11/11/2018 • 4 minutes to read • [Edit Online](#)

*Architectural guidance for developing adaptable, maintainable, and testable Xamarin.Forms enterprise applications*



This eBook provides guidance on how to implement the Model-View-ViewModel (MVVM) pattern, dependency injection, navigation, validation, and configuration management, while maintaining loose coupling. In addition, there's also guidance on performing authentication and authorization with IdentityServer, accessing data from containerized microservices, and unit testing.

## Preface

This chapter explains the purpose and scope of the guide, and who it's aimed at.

## Introduction

Developers of enterprise apps face several challenges that can alter the architecture of the app during development. Therefore, it's important to build an app so that it can be modified or extended over time. Designing for such adaptability can be difficult, but typically involves partitioning an app into discrete, loosely coupled components that can be easily integrated together into an app.

## MVVM

The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI). Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an app.

## Dependency Injection

Dependency injection enables decoupling of concrete types from the code that depends on these types. It typically uses a container that holds a list of registrations and mappings between interfaces and abstract types, and the concrete types that implement or extend these types.

Dependency injection containers reduce the coupling between objects by providing a facility to instantiate class instances and manage their lifetime based on the configuration of the container. During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first.

# Communicating Between Loosely Coupled Components

The `Xamarin.Forms` `MessagingCenter` class implements the publish-subscribe pattern, allowing message-based communication between components that are inconvenient to link by object and type references. This mechanism allows publishers and subscribers to communicate without having a reference to each other, helping to reduce dependencies between components, while also allowing components to be independently developed and tested.

## Navigation

Xamarin.Forms includes support for page navigation, which typically results from the user's interaction with the UI, or from the app itself, as a result of internal logic-driven state changes. However, navigation can be complex to implement in apps that use the MVVM pattern.

This chapter presents a `NavigationService` class, which is used to perform view model-first navigation from view models. Placing navigation logic in view model classes means that the logic can be exercised through automated tests. In addition, the view model can then implement logic to control navigation to ensure that certain business rules are enforced.

## Validation

Any app that accepts input from users should ensure that the input is valid. Without validation, a user can supply data that causes the app to fail. Validation enforces business rules, and prevents an attacker from injecting malicious data.

In the context of the Model-View-ViewModel (MVVM) pattern, a view model or model will often be required to perform data validation and signal any validation errors to the view so that the user can correct them.

## Configuration Management

Settings allow the separation of data that configures the behavior of an app from the code, allowing the behavior to be changed without rebuilding the app. App settings are data that an app creates and manages, and user settings are the customizable settings of an app that affect the behavior of the app and don't require frequent re-adjustment.

## Containerized Microservices

Microservices offer an approach to application development and deployment that's suited to the agility, scale, and reliability requirements of modern cloud applications. One of the main advantages of microservices is that they can be scaled-out independently, which means that a specific functional area can be scaled that requires more processing power or network bandwidth to support demand, without unnecessarily scaling areas of the application that are not experiencing increased demand.

## Authentication and Authorization

There are many approaches to integrating authentication and authorization into a Xamarin.Forms app that communicates with an ASP.NET MVC web application. Here, authentication and authorization are performed with a containerized identity microservice that uses IdentityServer 4. IdentityServer is an open source OpenID Connect and OAuth 2.0 framework for ASP.NET Core that integrates with ASP.NET Core Identity to perform bearer token authentication.

## Accessing Remote Data

Many modern web-based solutions make use of web services, hosted by web servers, to provide functionality for remote client applications. The operations that a web service exposes constitute a web API, and client apps should

be able to utilize the web API without knowing how the data or operations that the API exposes are implemented.

## Unit Testing

Testing models and view models from MVVM applications is identical to testing any other classes, and the same tools and techniques can be used. However, there are some patterns that are typical to model and view model classes, that can benefit from specific unit testing techniques.

## Feedback

This project has a community site, on which you can post questions, and provide feedback. The community site is located on [GitHub](#). Alternatively, feedback about the eBook can be emailed to [dotnet-architecture-ebooks-feedback@service.microsoft.com](mailto:dotnet-architecture-ebooks-feedback@service.microsoft.com).

## Related Links

- [Download eBook \(2Mb PDF\)](#)
- [eShopOnContainers \(GitHub\) \(sample\)](#)

# SkiaSharp Graphics in Xamarin.Forms

10/3/2018 • 2 minutes to read • [Edit Online](#)

*Use SkiaSharp for 2D graphics in your Xamarin.Forms applications*

SkiaSharp is a 2D graphics system for .NET and C# powered by the open-source Skia graphics engine that is used extensively in Google products. You can use SkiaSharp in your Xamarin.Forms applications to draw 2D vector graphics, bitmaps, and text. See the [2D Drawing](#) guide for more general information about the SkiaSharp library and some other tutorials.

This guide assumes that you are familiar with Xamarin.Forms programming.

## Webinar: SkiaSharp for Xamarin.Forms

## SkiaSharp Preliminaries

SkiaSharp for Xamarin.Forms is packaged as a NuGet package. After you've created a Xamarin.Forms solution in Visual Studio or Visual Studio for Mac, you can use the NuGet package manager to search for the **SkiaSharp.Views.Forms** package and add it to your solution. If you check the **References** section of each project after adding SkiaSharp, you can see that various **SkiaSharp** libraries have been added to each of the projects in the solution.

If your Xamarin.Forms application targets iOS, use the project properties page to change the minimum deployment target to iOS 8.0.

In any C# page that uses SkiaSharp you'll want to include a `using` directive for the `SkiaSharp` namespace, which encompasses all the SkiaSharp classes, structures, and enumerations that you'll use in your graphics programming. You'll also want a `using` directive for the `SkiaSharp.Views.Forms` namespace for the classes specific to Xamarin.Forms. This is a much smaller namespace, with the most important class being `SKCanvasView`. This class derives from the Xamarin.Forms `View` class and hosts your SkiaSharp graphics output.

### IMPORTANT

The `SkiaSharp.Views.Forms` namespace also contains an `SKGLView` class that derives from `View` but uses OpenGL for rendering graphics. For purposes of simplicity, this guide restricts itself to `SKCanvasView`, but using `SKGLView` instead is quite similar.

## SkiaSharp Drawing Basics

Some of the simplest graphics figures you can draw with SkiaSharp are circles, ovals, and rectangles. In displaying these figures, you will learn about SkiaSharp coordinates, sizes, and colors. The display of text and bitmaps is more complex, but these articles also introduce those techniques.

## SkiaSharp Lines and Paths

A graphics path is a series of connected straight lines and curves. Paths can be stroked, filled, or both. This article encompasses many aspects of line drawing, including stroke ends and joins, and dashed and dotted lines, but stops short of curve geometries.

## SkiaSharp Transforms

Transforms allow graphics objects to be uniformly translated, scaled, rotated, or skewed. This article also shows how you can use a standard 3-by-3 transform matrix for creating non-affine transforms and applying transforms to paths.

## SkiaSharp Curves and Paths

The exploration of paths continues with adding curves to a path objects, and exploiting other powerful path features. You'll see how you can specify an entire path in a concise text string, how to use path effects, and how to dig into path internals.

## SkiaSharp Bitmaps

Bitmaps are rectangular arrays of bits corresponding to the pixels of a display device. This series of articles shows how to load, save, display, create, draw on, animate, and access the bits of SkiaSharp bitmaps.

## SkiaSharp Effects

Effects are properties that alter the normal display of graphics, including linear and circular gradients, bitmap tiling, blend modes, blur, and others.

## Related Links

- [SkiaSharp APIs](#)
- [SkiaSharpFormsDemos \(sample\)](#)
- [SkiaSharp with Xamarin.Forms Webinar \(video\)](#)